[redacted]

UIUCDCS-R-72-504

AN INTERACTIVE SYNTAX ANALYZER

by

Wayne C. Sanford

January, 1972

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-72-504

AN INTERACTIVE SYNTAX ANALYZER

BY

Wayne C. Sanford

1972

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was supported in part by the National Science Foundation Grant No. US NSF-GJ-328.

ACKNOWLEDGMENT

The author would like to thank Professor J. R. Phillips for the advice and guidance which made this project possible, Mr. M. Ozga for his assistance in using the TWST65 compiler-compiler, and the Center for Advanced Computation and the Department of Computer Science for the use of their resources.

TABLE OF CONTENTS

Page

LIST OF FIGURES

1. INTRODUCTION

This project is motivated by the problems which face anyone starting to program in an unfamiliar language. The new user is usually not very familiar with either the syntax or the power of the language, but he needs to write programs to discover its capabilities. Surely his anxieties would be lessened if he could seat himself at a remote terminal and have each statement immediately analyzed for proper syntactic content, especially if a concerted effort is made to fully describe his errors to him.

A program, OL2/PARSER, has been developed which allows users of OL/2, a compact, powerful array processing language, to type their programs at a terminal and receive a statement by statement syntactic analysis for the purpose of determining whether their programs are syntactically correct. The program does not enable the user to construct a file which may be passed to an OL/2 compiler only because OL2/PARSER and OL/2 are implemented on separate machines; however, OL2/PARSER in its present state has significant teaching value, and continued development could be of further value to the OL/2 project. OL2/PARSER is maintained using TWST65 (1, 2), a syntax-directed compiler-compiler in use on the Burroughs 6500 computer system.

It is the purpose of this paper to show that syntax-directed, recursive-descent compiler-compilers can benefit the software designer and user in more ways than just providing a convenient method of implementing new languages. The paper will show how a representative compiler-compiler can be used to create an interactive syntax analyzer which is of some valu at its present stage of development and which shows promise of greater usefulness. The techniques and applications discussed are by no means specific to this project and may be used by any language development project, especially one utilizing syntax-directed compiling.

The core of the specific problem is that of identifying and correcting the users' syntactical errors. LaFrance (3) specifies syntactical errors as the third of five levels of errors which prevent a program from running successfully; clearl some effort should be made to enable the programmer to overcome these errors as easily as possible. LaFrance has developed an algorithm to correct syntactical errors at compilation; this project enables the user to correct syntactical errors as he creates his source program.

Usually it is the task of the compiler to detect and identify syntactic errors. Most of the effort in compiler writing is not devoted to identifying and describing syntactical errors, and most compilers scan to some easily found symbol afte an error before resuming compilation. The result can often be a long series of errors traceable to one previous mistake. The approach of this project allows each syntactic error to be handled separately, with little or no effect upon succeeding statements. Also, since it is devoted to syntactical errors, a great deal of attention is paid to error descriptions.

The approach of this project does not provide as thorough an analysis of a user's program as a quick-and-dirty compiler might, but it is certainly easier to code and offers the user the opportunity to correct his errors as they are found.

Overview of the Thesis

The next chapter discusses how syntactic errors can be determined using recursive-descent parsing by inserting error routines into the parsing tree. The difficulty of locating errors with respect to both the syntax and the input string are described.

Chapter 3 discusses the potential value of a syntax analyzer, especially an interactive syntax analyzer, to both the language designer and the language user. The coding and use of such a program offers the designer a rapid and simple means to analyze proposed and alternate syntactical constructions before incorporating them into a compiler, where undiscovered syntactic ambiguities may later cause severe difficulties. Such a program can enhance users' understanding of the language, and should provide more rapid turnaround for debugging.

Chapter 4 briefly explores other directions which such a project might take. The discussion of finer error location, error correction, semantic checking, teaching applications, and file editing and building capabilities are directed toward the OL/2 project, but are by no means specific.

The conclusion is that imaginative use of compiler-compilers can be of significant value in the development of new software.

2. METHOD OF ERROR DETECTION

The central problem in constructing a syntax analyzer is developing a method of finding and describing syntax errors. The method used in OL2/PARSER, with fair success, is that of inserting error routines as alternatives within the syntax of the language. Error messages are in terms of the nonterminals of the syntax, and the series of messages related to an error describe the path taken by the syntax analyzer through the parsing tree of the language to the point of the error. This chapter will discuss that method.

2.1 Parsing Trees

Throughout this paper, reference will be made to the parsing tree of a language. The parsing tree of a language is the most inclusive tree in the forest of trees which can be constructed from the Backus Naur Form or modified Backus Naur Form definition of the syntax of that language. The reader is assumed to be familiar with the basic concepts of trees and the composition of BNF.

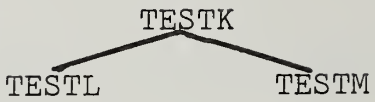
The definition of language syntax in BNF is not uncommon; for example, ALGOL is usually specified in BNF. Compiler-compiler often utilize a streamlined form of BNF as input (1, 4), and Trout shows that TBNF, which is used in TWST65, is as powerful as pure BNF (1). Appendix A contains a brief description of some of the symbols of TBNF (Figure A.1), some of which will be used in this paper.

The process of forming the parsing forest of a language is that of forming a parsing tree for each nonterminal defined in the language. This is done in a very straightforward manner. Nonterminals appearing to the left of the "::=" operaʋor are interpreted as the root of the parsing tree associated with that nonterminal. Alternative definitions are interpreted as separate branches of the root. Nonterminals and terminals on the right hand side of the "::=" operator are interpreted as nodes at successive levels of the parsing tree.

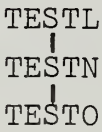
The node representing a particular nonterminal or terminal is labeled "TESTelement name," for example, TESTK at the top of Figure 1 on the next page. This convention is used because the parsing algorithm will "interrogate" nodes and expect either TRUE or FALSE to be returned. A node will return the value TRUE when it is "interrogated" if the nonterminal or terminal to which it corresponds exists in the string being scanned starting at the scanner location when the node is "interrogated." Otherwise, the value FALSE will be returned.

The examples in Figure 1 show how parsing trees are constructed from BNF productions. <K> is defined as either <L> or <M>; so TESTK has two branches, TESTL and TESTM. <L> is defined as <N> followed by <O>; so TESTL has one branch with TESTN followed by TESTO. <M> has three alternate definitions, so TESTM has three branches. The definition of <N> shows how square brackets may be used to simplify syntactic definitions. <O> is defined as <*I>, a symbol from TBNF, which accepts strings of alphanumeric characters with an initial alphabetic character.

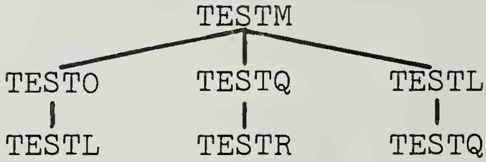
```
<K> ::= <L> / <M> ;
```



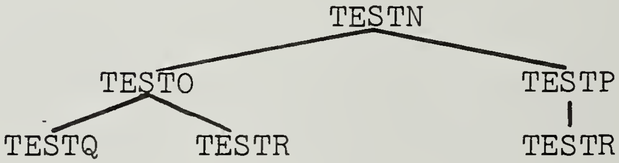
```
<L> ::= <N><O> ;
```



```
<M> ::= <O><L> / <Q><R> / <L><Q> ;
```



```
<N> ::= <O> [<Q> / <R> ] / <P><R> ;
```



```
<O> ::= <*I> ;
```

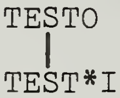


Figure 1. Syntax definitions and parsing trees

The set of trees formed from all the nonterminal definitions in a language is the parsing forest associated with that language. The parsing tree for the language can be considered to be that tree which has the most inclusive nonterminal in the language as its root. All of its nodes are parsing trees in the parsing forest. In the example of Figure 1, <K> is the most inclusive nonterminal. Most language definitions would use <PROGRAM> or something similar. (For the purpose of constructing a syntax analyzer as described in this paper, the syntactic unit at the statement level should be used to define the parsing tree of the language, for the method being developed is primarily directed toward interactive parsing at that level (Chapter 2.4).) The parsing algorithm must determine whether the parsing tree for the language has the value TRUE or the value FALSE.

The parsing algorithm traverses any parsing tree by interrogating nodes in the tree. It interrogates a node by saving information on a stack for continuing the parse after the interrogated node returns a value and traversing the parsing tree of the interrogated node. The process of stacking and interrogating continues until some interrogated node looks for particular terminal symbols, at which time the input string is scanned for the required symbols starting at the last character successfully scanned, and a value, TRUE or FALSE, is returned.

The parsing algorithm traverses a parsing tree by interrogating its nodes, <u>other</u> than the root, in the following order. Start with the node at the first level of the left-most branch. If a node returns the value TRUE, interrogate the left-most son of that node, except that if that node is a leaf of the tree, return the value TRUE for the tree being traversed. If a node returns the value FALSE, interrogate the brother to the right of that node. If the node has no brothers to its right, backtrack by ascending one level at a time until either a brother to the right is found, in which case interrogate that brother, or all branches from the root have been examined (there are no more brothers to the right), in which case return the value FALSE for the tree being traversed. Note that every time the algorithm is forced to ascend a level, it must re-scan some previously reduced portion of the input string.

The parse of any tree is said to be complete if a leaf returns the value TRUE, since all nodes between the root and that leaf must have been interrogated with a resulting TRUE return. The parse is incomplete if the algorithm exhausts the alternatives and returns the value FALSE. The parsing algorithm always starts in the parsing tree of the language; if it is complete, a syntactically correct <STATEMENT> or <PROGRAM> has been reduced. The parsing tree for the language will be incomplete if a syntax error causes the parsing algorithm to unsuccessfully exhaust all alternatives within it.

For example, in Figure 1, TESTK will be TRUE if either TESTL or TESTM returns the value TRUE and TESTL will be TRUE only if TESTN and TESTO return TRUE when interrogated in that order. As as example of traversal, consider the path taken by the parsing algorithm in reducing the following valid sequence of nonterminals.

<O><P><R><O>

TESTK interrogates TESTL. TESTL interrogates TESTN. TESTN interrogates TESTO, which returns TRUE. TESTN then int: rogates TESTQ and TESTR, which both return FALSE. TESTN backtracks, so that <O> is no longer reduced, and interrogates TESTP. TESTP returns FALSE; so TESTN must return FALSE; so TESTL must return FALSE. TESTK now interrogates TESTM. TESTM interrogates TESTO, which returns TRUE, and then TESTL. TESTL interrogates TESTN, which interrogates TESTP after TESTO returns FALSE. TESTP and TESTR return TRUE, so TESTN returns TRUE, and TESTL interrogates TESTO. TESTO returns TRUE, TESTL returns TRUE, TESTM returns TRUE, and the parsing tree TESTK is complete.

The algorithm has done a great amount of work to reduce a string containing four nonterminals. The algorithm would have done even more work in determining that the sequence

<O><P><R>

is not a valid instance of <K>. After <O>, <P>, and <R> have been reduced as above, TESTL would interrogate TESTO, which would fail. The algorithm would be forced to backtrack and examine the final two branches of TESTM before TESTM would finally return the value FALSE.

In the second example considered, clearly the user should be informed that he has failed to include <O> following the occurrence of <R> in his string. The next section of this chapter will explain how parsing trees can be modified so that such error routines may be inserted as alternatives within the syntax.

2.2 Insertion of Error Routines

In a parsing tree for a language as defined in the previous section, a complete parse is associated with a syntactically correct statement, and an incomplete parse indicates that a syntactic error is imbedded in the input string. What can be done beyond telling the user that an error exists in his input to the parser? How specifically can the error be described to the user?

The solution to these problems depends on having unambiguous parsing trees and knowing the location of the parsing algorithm when it exhausts its possible alternatives.

An unambiguous parsing tree is one which has the property that, if a node at any level of the tree returns the value FALSE, the parsing algorithm need only examine the brothers of that node and their descendants before returning an answer. In other words, an unambiguous parsing tree is one in which backtracking above the node currently being tested is of no value since all subsequent alternatives will fail.

An ambiguous parsing tree is one which has the property that, if a node at any level returns the value FALSE, the parsing algorithm must examine all remaining nodes at the current level on the current branch, their descendants, and all nodes on any remaining branches before it can return an answer. In other words, an ambiguous parsing tree is one in which the knowledge of what has been previously reduced is of no value. The syntax of Figure 1, and its associated trees, is clearly ambiguous, as shown in the previous section. In the example at the end of the preceding section, the fact that TESTO returned the value TRUE while TESTL was first being interrogated was not useful since TESTM contains a branch with TESTO as the first node.

For the ambiguous parsing tree in Figure 2.a on the next page, if TESTJ returns the value FALSE, the parsing algorithm would be forced to interrogate TESTE, TESTD, and TESTB (on the right branch), TESTF, TESTG, TESTH, and TESTI before returning the value FALSE for TESTA if the user intended to use the <J> construct in his statement and committed an error. In fact, the parsing algorithm need not have interrogated any further nodes, since TESTC had already returned the value TRUE (otherwise the algorithm could not interrogate TESTJ). Only <J> can follow <C> in the syntax represented.

The unambiguous tree shown in Figure 2.b allows the algorithm to stop parsing and issue an error message if TESTJ returns the value FALSE since there is no other path through the tree which begins with TESTB. The message can be that an incorrect occurrence of <J> follows an occurrence of <C>.

If the parsing tree for <J>, TESTJ, is also unambiguous, then a similar message involving the nonterminals or terminals used in the definition of <J> can be produced where the parse failed. The process can continue until the parsing tree for the last node interrogated,which will halt when some incorrect terminal symbol is found, produces the most specific message possible.

```
<A> ::= <B> [<C><J> / <E>] / <D> /
            <B><F> [<G> / <H> / <I>] ;
```

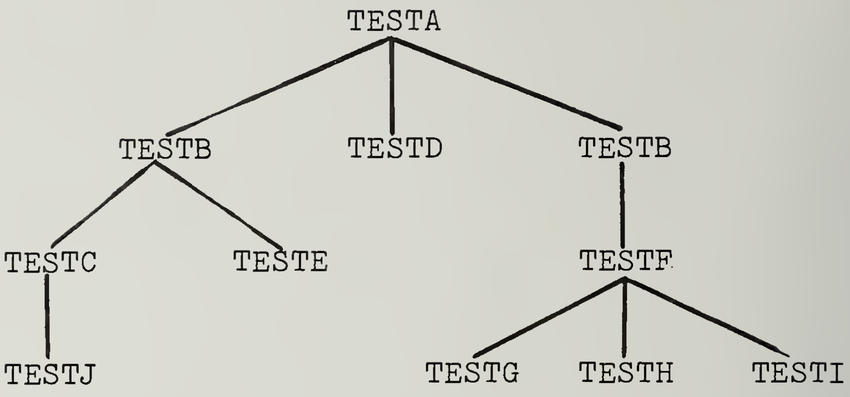


Figure 2.a. Ambiguous parsing tree

```
<A> ::= <B> [<C><J> / <E> /
                 <F> [<G> / <H> / <I>] ] /
            <D> ;
```

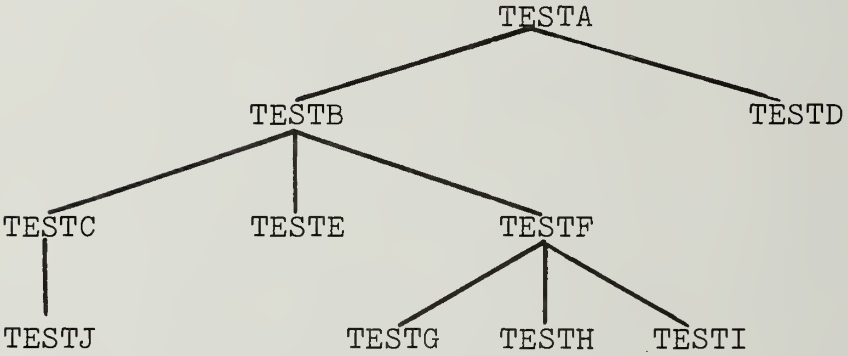


Figure 2.b. Unambiguous parsing tree

Figure 2. Ambiguous and unambiguous parsing trees

The syntax of a language will usually contain words and symbols which can serve to make the parsing forest associated with the language unambiguous, or at least make the trees associated with some of the nonterminals unambiguous. For example, in OL/2, a <PARTITIONSTATEMENT> must start with the word "PARTITION". Once the word "PARTITION" has been recognized, the parsing is limited to the tree TESTPARTITIONSTATEMENT. If the word "PARTITION" is found and TESTPARTITIONSTATEMENT returns the value FALSE, there is absolutely no reason to interrogate the nodes for the remaining statement types in the parsing tree for OL/2, since no other statement type starts with "PARTITION". Careful examination of the syntax of OL/2 shows that each of the twelve types of <OL2STATEMENT> contains a unique symbol near the beginning of the statement type (Appendix A).

The key to locating syntactical errors with respect to the parsing tree of a language with the approach of this paper is finding signpost symbols which identify the user's intention to use a particular syntactic construction. The single most serious drawback to this system is that errors involving the signpost symbols will prevent the parsing algorithm from entering those sections of the parse from which the best messages can be supplied and may force the algorithm into an incorrect parsing tree. For example, "LET" is the signpost symbol for <NEWOL2BLOCK> and "SET" is the signpost symbol for <SETSTATEMENT> in OL/2. The use of "LET" instead of "SET" will cause OL2/PARSER to issue error messages pertaining to <NEWOL2BLOCK>, and the use of "SAT" instead of "SET" will force OL2/PARSER to consider the string an <ASSIGNMENTSTATEMENT> since "SET" and "LET" are recognized as signposts but "SAT" is considered a variable name.

Once the signpost symbols in each nonterminal definition have been located, error routines can be inserted as additional alternatives in portions of the syntax following a signpost. In the example of Figure 2.b, the parse can be modified so that if TESTB is TRUE and TESTC is TRUE the parsing algorithm will expect to find an occurrence of <J>, since TESTC acts as a signpost. If TESTC is TRUE, <J> must follow or a message informing the user that <C> has been reduced but <J> does not follow in an intended occurrence of <A> should be produced.

Figure 3 on the next page shows how the unambiguous syntax of Figure 2 can be modified by the inclusion of error routines. The square brackets of TBNF are used to indicate how error routines are inserted in syntax using that form of BNF. The equivalent parsing tree appears below the definition. Failure of TESTD cannot be used to precipitate an error routine at this level, since the parsing tree from which TESTA is being interrogated might not be unambiguous. If it is unambiguous, the error routine following TESTA will be entered after TESTA returns the value FALSE. Failure of TESTF, however, can cause ERRORCEF to execute since TESTB must be TRUE, a signpost that the user intended to use <A> in his entry.

The fact that an error exists and can be limited to the set of currently interrogated nodes in an unambiguous parsing

```
<A> ::= <B> [ [<C> [<J> / <ERRORJ>] /
                <E> /
              <F> [<G> / <H> / <I> / <ERRORGHI>] /
            <ERRORCEF>] / <D> ;
```

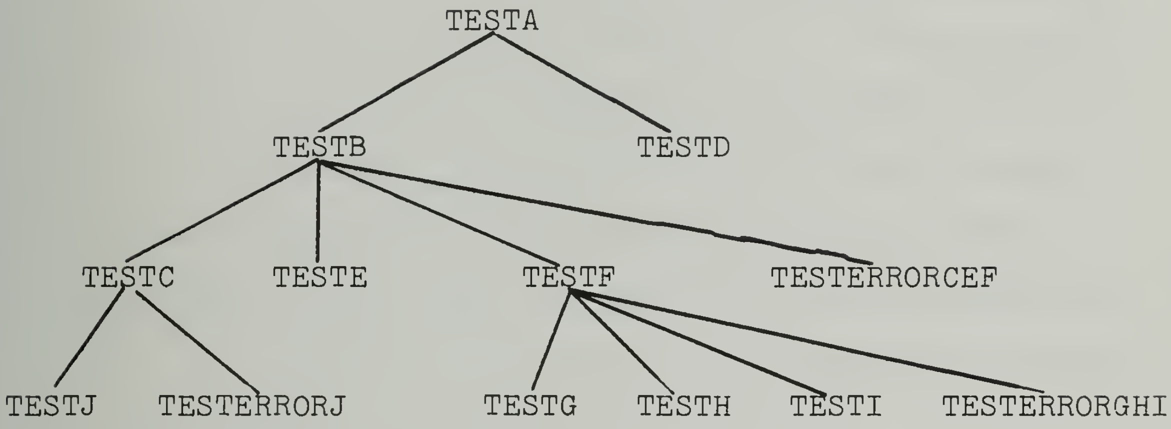


Figure 3. Insertion of error routines into unambiguous trees

structure allows the supression of all backtracking as defined in the order of traversal (Chapter 2.1). The value FALSE will be assigned to each interrogated node, causing the parsing algorithm to empty its stack. Consequently, the discovery of an error will cause the parse of the input string to be incomplete. The user will know that there is an error in his input string, the error routines will describe the error to him in terms of the syntax, but only the one error will be found and described. Only the use of error correcting routines (3) can enable the parsing algorithm, in general, to continue. The method of this thesis is an attempt to positively identify syntax errors. It is the opinion of the author that it is better to give the user as much information as possible about the first error in a statement and stop parsing than to describe the first error, attempt recovery, and take the chance of issuing misleading error messages.

One advantage of the use of a stack in controlling the parsing algorithm is that one message can be displayed for each node which returns a value of FALSE as the algorithm is forced back through a series of interrogated nodes. If the nonterminal name associated with each such node is displayed along-side the message the node produces, the user can realize how his input was parsed. The user who refers to a listing of the syntactic definitions of the language he is using can then trace the action of the parser through his string if he does not understand the error. The syntax listing will thus show him how the parsing mechanism reaches its decisions and what it expects.

Using the method of this paper, errors may only be described in terms of the nonterminals of the syntax. Errors must be described as a missing syntactic unit following the occurrence of a previously reduced syntactic unit as defined by a more inclusive syntactic unit. The problem is complicated by the existence of large numbers of alternatives in the syntax. In the syntax of Figure 2.b, if the user entered a string in which he intended to use <G>, and if no signpost for <G> were reduced by the syntax analyzer before an error is found, the most accurate error message would be that of ERRORGHI, namely that an incorrect occurrence of either <G>, <H>, or <I> follows the occurrence of <F> as defined by <A>. Similarly, errors concerning very large syntax definitions may be vague, such as "INCORRECT <ARITHMETICRIGHTHANDSIDE> FOLLOWING '='", which really says very little. The user must be informed that uninformative error messages are usually due to incorrect signposts.

2.3 Locating Errors in the String

Explaining to the user exactly where in the input string an error occurs is probably impossible, since an error may only force the parse down the wrong branch of some parsing tree, in which case the parse may not fail until it has successfully reduced some portion of the input string. Consider what happens when a left square bracket, "[", is used instead of a left parenthesis, "(". The parse will treat the "[" as a signpost. The parse will expect a matching right square bracket. If square brackets and parentheses are used to enclose similar syntactic units, the parse will not fail until it reaches the right parenthesis. The syntactic error may not always occur simultaneously with the user's actual error.

It is not difficult, however, to indicate a portion of the input string which includes the error. The parsing algorithm generated by a compiler-compiler must maintain a pointer to the input string which indicates which character must be scanned once the previous character is reduced. The value of this pointer is easily displayed as a marker under the input string at the appropriate position.

2.4 Interactive Considerations

The method of locating errors discussed so far applies to syntax analysis at any level of syntax. It is expected that the actual syntax analyzer will be written using a syntax directe compiler-compiler, since the method involves modifying the syntax Making the syntax analyzer interactive is a simple process on a machine which supports remote terminals, since the software for terminal input/output is usually made transparent to users. Usually all that is required is proper file attribute declaration.

Interactive syntax analysis does warrant special considderation, however. The user's entries will normally be limited to one line at a time, approximately the size of a statement in most languages. The user will obtain the most benefit if the analysis of his entries is carried out at the statement level, since that will allow him to catch and correct his errors almost as soon as he makes them. If the results of the complete or incomplete syntax analysis are not displayed at the conclusion of each statement, the syntax analyzer will not seem much different from a keypunch. In addition, since the sugges ed method causes parsing failure back to and including the largest syntactic unit defined, it makes sense to keep that unit at the same level as the input in order to easily reinitialize the parsing algorithm after an error has been found.

2.5 Examples from OL2/PARSER

A few examples from OL2/PARSER should help to clarify the concepts mentioned. The following figure shows some incorrect OL/2 statements (underlined), an error position marker, and appropriate error messages. (Appendix B documents the syntax of OL2/PARSER.)

```
LET X BE A FINITE VECTOR SPACE OF DIMENSION (N);
                        <

<DEFINITION>:  'DIMENSIONAL' MISSING FOLLOWING 'FINITE'
<NEW OL2BLOCK>:  INCORRECT <DEFINITION>


LET K BE A FIXED SCALAR
                <

<NEW OL2BLOCK>:  INCORRECT <DEFINITION>


X(K) = N***K ;
           <

<OL2FACTOR>:  INCORRECT <OL2EXTRA> FOLLOWING '**'
<ASSIGNMENTSTATEMENT>:  INCORRECT <COMPAREEXPRESSION>
```

Figure 4. Examples from OL2/PARSER

In the first example, the user had not included the word "DIMENSIONAL" after "FINITE," as required by the syntax of OL/2 (Appendix A). The error position marker, "<", is displayed under the first character not scanned at the time of the error, in this case, the blank between "VECTOR" and "SPACE", since the word "VECTOR" is scanned as a single unit instead of a sequence of individual characters. The scanning mechanism scans "VECTOR" from the string, the parsing algorithm compares that unit to the terminal word "DIMENSIONAL", and, since the signpost "FINITE" had previously been recognized, the first message is displayed. The message shows that TESTDEFINITION is the node being interrogated at the time the error is found. The second message indicates that TESTDEFINITION is being interrogated from TESTNEWOL2-BLOCK. The second message is produced because "LET" is a signpost for <NEWOL2BLOCK> and because the nodes TESTIDLIST and TESTDENOTATIVEPHRASE preceding TESTDEFINITION have been satisfied by "X" and "BE A" respectively. The feature of displaying additional messages as the stack empties is most helpful when errors occur within parentheses, square brackets, or angle brackets, where TESTPL1EXPRESSION often occurs. For nonterminals such as <PL1EXPRESSION> which are used in many places in the syntax, knowing from where their parsing trees are interrogated may often be of value.

The second example shows that the error messages cannot always be as specific as the first message in the prior example. The algorithm interrogates TESTDEFINITION but can locate no signpost within TESTDEFINITION ("VECTOR" or <CLASS>) before it tries to reduce "FIXED". The only remaining alternatives are the words "SCALARS" and "SCALAR", so TESTDEFINITION is g /en the value FALSE, and the error message is forced. The presence of "LET" prevents the algorithm from interrogating the parsing tree of any other statement type.

The third example shows error messages from another statement type, <ASSIGNMENTSTATEMENT>. The first message indicates that "*" is not a legal operand for the exponentiation operator. The character "K" is not scanned from the string at the time this message is produced. The second message indicates that the parsing algorithm reached this point by interrogating TESTASSIGNMENTSTATEMENT and TESTCOMPAREEXPRESSION.

The second example also shows that errors subsequent to the first error in the string are not found by an algorithm which is forced to fail at all levels by the occurrence of an error. There is no terminal semicolon at the end of the statement, but the syntax analyzer cannot find the error since it re-initializes itself and asks for more input after displaying a set of error messages.

3. APPLICATIONS OF AN INTERACTIVE SYNTAX ANALYZER

3.1 For the Language Designer

This chapter of the thesis will explain of what value constructing a syntax analyzer of the complexity of OL2/PARSER, incorporating the methods of the preceding section, can be to the language development project and the language user. This subsection is concerned with the development of the language; the next involves the benefits to users.

If a design group develops a syntax analyzer as the first step of their project, they can benefit in two ways. First, theywill be able to write the version of their syntax which will be most efficiently handled by their compiler. Second, once they have the syntax analyzer, they can use it to preprocess user programs, obviating the need for syntax analysis in the compiler.

3.1.1 Developing Syntax

The group which intends to use a compiler-compiler to write the compiler for their language clearly stands to gain insight into how their compiler will operate by building a syntax analyzer using the same compiler-compiler. Developing a comprehensive set of syntactic error messages will necessitate the understanding of the string handling and reduction techniques employed by the compiler-compiler and the discovery of the signpost symbols in the proposed syntax.

The developers should strive to make the parsing trees for their syntax as unambiguous as possible. This will enable them to produce the most meaningful set of error messages, and will later allow the most precise placement of semantic routines. The first result, production of the most meaningful set of error messages, is evident from the preceding chapter. The second result, precise placement of semantic routines follows directly.

Semantic routines, which must maintain a symbol table and emit machine or intermediate code, should be executed as soon as possible after the reduction of their associated syntactic units so that they may be accomplished with a minimum of program linkage. Routines in an ambiguous syntax must be delayed until there is no chance of incorrectly processing the syntactic unit in question. In the ambiguous syntax shown in Figure 5, for example, semantic processing of <B> cannot be accomplished at the left-most son of the root because the parsing algorithm may be forced to examine its brother or even backtrack above that node. In either case, the previous reduction must be undone. There is no such problem in the unambiguous version of the syntax. Semantic processing as a function of <B> can be accomplished as soon as TESTB returns the value TRUE. Those semantic actions which depend on whether <C>, <E>, or <F> follows <B> must of course wait until one of those syntactic units is reduced.

The effort to develop an unambiguous parsing tree for the language will also allow the designer to uncover and correct inconsistencies and repetitions in the syntax. An example of inconsistency would be equivalent syntactic definitions of nonterminals intended to be distinct. Two alternatives with different semantic meanings might be equivalent, even though defined in terms of different nonterminals. In such a case, a string meant to contain an instance of the second structure would always be reduced as an instance of the first. This problem is most likely to arise when a large number of alternatives are defined in the syntax. The problem would not be evident to a person reading the syntax; however, someone attempting to produce separate error messages for each of the forms would find the parallel constructions when unable to precipitate error messages regarding the second construct.

Consider, for example, what would happen if <C>, <J>, and <E> of the syntax in Figure 5.b were defined in the following way.

```
<C> ::= <*I><semantic routine C> ;

<J> ::= (<*N>) ;

<E> ::= <SPECIALID><semantic routine E>(<*N>) ;

<SPECIALID> ::= ' <*I>' / <*I> ;
```

Clearly no string would ever be reduced by TESTE unless the apostrophes were in fact used to surround the identifier. This problem might not be discovered at the time the syntax is written, especially if this is only a small part of a large, complex grammar.

The language designer can also use the syntax analyzer for the language to discover whether strings he intends to be legal are in fact accepted by the syntax he has defined. He will be able to determine the completeness of his syntax.

3.1.2 Syntactic Preprocessing

The language project which has developed a syntax analyzer can use it as a preprocessor for the language compiler. An interactive analyzer such as OL2/PARSER, can be easily modified (Chapter 4) to build source files for the compiler which are free of syntactic errors. A similar analyzer constructed to handle complete files can ensure that all source files passed to the compiler are syntactically correct. If either users are required to build their source files using the interactive system or all source files are preprocessed by a syntax analyzer or some combination of the two previous ideas is used, syntactical error routines need not be included in the compiler. The insertion of semantic routines may be simplified by the knowledge that source input to the compiler will contain no syntactic errors.

3.2 For the Language User

The language user also stands to benefit from a syntax analyzer such as OL2/PARSER. He will receive faster turnaround, more thorough syntax checking with better error descriptions, and a better understanding of the language.

The user will receive faster turnaround while eliminating syntax errors if his files are preprocessed by a syntax analyzer because the syntax analyzer will use less memory than the compiler, allowing the analyzer to be scheduled more easily in a multi-programmed environment, and the syntax analyzer will be faster than the compiler in searching for syntax errors. The claims of smaller memory requirements and greater parsing speed are predicated on the fact that there are no complex semantic routines in the syntax parser to manipulate symbol tables or emit code.

He will receive more thorough syntax checking with better error descriptions because the function of the syntax analyzer is to find and explain syntactic errors, whereas the compiler is primarily concerned with generating executable code.

He will receive a better understanding of the language because his errors will be more clearly explained, and because their explanation will be in terms of the syntax. He will begin to realize how the compiler works if he studies his errors and will thus be in a position to more fully utilize the capabilities of the language.

An interactive syntax analyzer is especially useful since the user learns of his syntactic errors almost as soon as he makes them. An analyzer which works essentially line by line and terminates the parse after finding one error will describe those errors found independently of each other, so one error will not initiate a chain of error messages. The user can be certain that there is indeed an error when an error position marker and some messages are displayed.

4. EXTENSIONS TO AN INTERACTIVE SYNTAX ANALYZER

Syntax analyzers, as described up to this point, are fairly straightforward, are easy to code using a recursive-descent compiler-compiler, and have limited capabilities. Within the same framework, however, there are several other functions which might be incorporated into the syntax analyzer for a given language at the option of the language designer. The extensions which are feasible for all syntax analyzers include more accurate error location in the string, automatic error correction, semantic preprocessing, and global analysis; in addition, teaching and editing capabilities might be written into interactive syntax analyzers.

4.1 More Accurate Error Location

It was mentioned earlier that it may be impossible to define the initial incorrect character in the input string. The approach outlined in Section 2.3 is to show the user where the scanning of the input string stopped, which places the error in that portion of the string prior to that point. Incorporating one position marker indicating a point after which the error occurs is not possible in general because the intentions of the user are not known. The initial error may force the parsing algorithm into the wrong parsing tree; consequently, some of the error messages produced will be of no value to the user.

It would be possible, however, as the algorithm interrogates successive nodes, to save the position of the last character successfully scanned when a node is interrogated on the stack with whatever other information is necessary for backtracking through the parse. Then, as the errors pertaining to each unsuccessfully interrogated node are listed, a position marker could be displayed which would indicate that no error had been found at the time the particular node was interrogated. The user might be overwhelmed by pointers if an exceptionally long path through the parsing trees was taken, but the pointers still convey information about the parse up to the point of the error. Usually only two or three error messages are displayed under an error, so that the portion of the string containing the error will be made fairly clear.

4.2 Automatic Error Correction

LaFrance (3) has developed an algorithm for correcting syntactic errors based solely on the syntax of a language. His method was developed for syntax specifications using Floyd productions; however, he states that it could be adapted to recursive-descent parsing methods.

4.3 Semantic Preprocessing

Obviously, semantic routines can be included in the syntax analyzer written using a compiler-compiler. The more semantic routines included, the more like a compiler the syntax analyzer becomes. The syntax analyzer could be extended so far as to be a quick-and-dirty compiler with good error messages or, even farther, to be the language compiler. The insertion of a full complement of semantic routines into a structure such as defined in this paper, which contains error routines as alternatives in the syntax, would undoubtedly be a tricky problem. Also, the resulting program would become very large and unwieldy.

Although the objections to trying to do everything in one program are many, there are good reasons for putting limited semantic routines into the syntax analyzer. There are some semantic problems which are so easy to catch that including the routines necessary to find them would be simple. This group of problems includes insuring that identifiers are used consistently with their definitions and checking for proper block structure.

In OL/2, for example, variables may be declared as scalars, vectors, matrices, or higher dimensional operators. Operands cannot be arbitrarily combined in an expression because they may violate the rules of matrix algebra. Some of the possible errors in expressions can be classified as syntactic; but with dynamic arrays most errors would be semantic. A simple symbol table maintained by the syntax analyzer with a list of attributes would allow it to determine a more inclusive set of syntactic errors as well as some semantic errors similar to those mentioned above.

4.4 Global Syntax Analysis

Syntax analyzers as defined to this point in the paper can be considered "local" syntax analyzers since they are concerned only with detecting and describing errors at the statement level. Higher level, or "global", syntax analysis would involve errors at the block or program or other defined levels. A syntax analyzer could be programmed to continue ana_ysis at each higher unit level upon deciding that a new input string caused no error at the current level of analysis. The user would then discover after each statement whether it affects the syntactic structure of other levels of the program.

Alternatively, an analyzer could be programmed to examine a user's source file for "global" syntactic errors after the final statement is entered. This would not allow the user to realize the full effect of each statement as he types it, but it would allow him to worry about only one level of syntax at a time.

4.5 Teaching

An interactive syntax analyzer, such as OL2/PARSER, could easily be appended to a teaching routine for the purpose of evaluating student responses. The teaching program could be one that leads the student through the features of the language, or it could be programmed to respond to student questions about various features.

In either case, since the teaching routine would ask the student for particular types of statements, the parsing trees of the syntax analyzer could be modified to require inclusion of expected "signpost" symbols, thus enabling even better description of student errors than with the previously developed syntax analysis techniques.

4.6 File Editing and Building

An interactive syntax analyzer which is implemented on the same machine as the compiler for the language can easily build a file of the statements entered into it. A better process would be to build a file of syntactically correct statements entered into it; that file could then be passed to the compiler if the user desired. The most elegant possibility is that of providing editing features in the syntax analyzer.

With the capabilities of adding and deleting characters and lines from a disk or memory file available, the user could create a source file for the language, enter statements, correct syntactic errors as they are found, alter previously entered statements, and add and delete statements to and from the file. When the user is satisfied that he has constructed a complete program, he can pass it to the compiler and be certain that it is free of syntactic errors.

5. CONCLUSION

This paper has shown that the language development project using a recursive-descent compiler-compiler can use that vehicle to thoroughly test the syntax of its language, to separate the problems of handling syntactic errors from the problems of semantics, and to create a syntax analyzer that can be of significant value to the users of the language.

The compiler-compiler can be used to develop a syntax analysis algorithm which can locate syntactic errors and describe them with a fair degree of accuracy. The syntax analyzer can be coupled with as many other functions, including automatic error correction, semantic error detection, teaching, and file editing and building, as desired.

This approach is limited primarily by the limitations of syntax-directed compilers, which have not been discussed; however the approach, if used with imagination, can produce many worthwhile results.

LIST OF REFERENCES

(1) Trout, H. R. G., "A BNF Like Language for the Description of Syntax Directed Compilers," Report No. 300, Department of Computer Science, University of Illinois, Urbana, Illinois (January, 1969).

(2) Trout, H. R. G., "TWST Users' Manual" (preliminary draft), ILLIAC IV Project, University of Illinois, Urbana, Illinois (January, 1971).

(3) LaFrance, J. E., "Syntax-Directed Error Recovery for Compilers," Report No. 459, Department of Computer Science, University of Illinois, Urbana, Illinois (June, 1971).

(4) Gaffney, J. L., Jr., "TACOS: A Table Driven Compiler-Compiler System," Report No. 325, Department of Computer Science, University of Illinois, Urbana, Illinois (June, 1971).

(5) Abel, N. E., "The Little Golden Book of the B6500," ILLIAC IV Project, University of Illinois, Urbana, Illinois.

APPENDIX A

DOCUMENTATION OF OL2/SYNTAX

OL/2 is currently implemented only on the IBM SYSTEM/360 MODEL 75 in the Digital Computer Laboratory, where it is maintained using TACOS (4), a PL/I based compiler-compiler. OL2/PARSER, an interactive syntax analyzer for OL/2, is implemented on the B6500 currently in the Coordinated Science Laboratory. It is maintained using TWST65 (1, 2), an ALGOL based compiler-compiler, as documented in Appendix D.

The file OL2/SYNTAX defines the syntax of OL/2 in the metalanguage of TWST65, a modified form of BNF called TBNF, as described by Trout (1, 2). It is constructed directly from the IBNF listing for the generation of OL/2 by TACOS dated May 1, 1971. It is as consistent as possible with the IBNF definition; although the norm declaration feature, which is not defined, is omitted entirely, and the definition of <OL2/LEFTHANDSIDE> is somewhat simplified. Any differences in character use are due to the different character sets on the two machines, and to special characteristics of some symbols on the B6500.

For example, the characters '[' and ']' are used in place of '|_' and '_|' respectively. Also, the symbol '#' is used instead of '%' to surround PL/I statements and the '"' symbols around 'NULL' in <COMPAREEXPRESSION> are not used because the characters '%' and '"' have special string handling meaning to the B6500 system.

Figure A.1 defines the reserved symbols used by TWST65. The usage of most of these symbols should be evident from their usage in OL2/SYNTAX; the following examples should eliminate any confusion.

Usage of '%'

In the definition of <LINEARITY>, 'CONCATENATION ALLOWED HERE' is a comment since it follows '%'. TWST65 ignores everything following a '%' character in a line.

Usage of '#'

In the definitions of <CLASS> and <ONORINASPACE>, 'OPERATOR' and 'ON' are defined as terminal symbols since they follow a '#' character. 'OPERATOR' and 'ON' are normally reserved words.

Usage of <ANY> and BUT

The definition of <PL1STATEMENT> uses <ANY>, BUT, '#', and '*' to cause any string of characters surrounded by '#' symbols to be accepted as a <PL1STATEMENT>.

OL2/SYNTAX is maintained as described in Appendix D; a listing follows Figure A.1.

TABLE OF TWST65 RESERVED SYMBOLS USED IN OL2/SYNTAX

TWST65 SYMBOL	MEANING
< >	'Angle' brackets used to surround non-terminal names
::=	Is defined as
/	Separates alternatives within definition
;	Terminates definition
?	Optional element <A>? ::= <A> / empty;
*	Kleene star <A>* ::= empty / <A><A>* ;
[ ]	'Square' brackets used to delimit groups of elements to be treated as a unit
%	Stop scanning this line
#	Use following reserved symbol as terminal symbol
LIST	LIST <A> ::= <A><A>* ;
SEPARATOR	LIST <A> SEPARATOR <B> ::= <A> [<B><A>]* ;
<*I>	Identifier scanning atom
<*N>	Unsigned integer scanning atom
<*R>	Unsigned real number scanning atom
<ANY>	Any character
BUT	Modifies preceding construct to exclude following construct

Figure A.1. Table of TWST65 symbols

OL2 SYNTAX AS DEFINED FOR THE OL2 INTERACTIVE SYNTAX PARSER

THE SYNTAX IS WRITTEN IN TBNF (TRANSLATABLE BACKUS NAUR FORM) AS DESCRIBED IN DCS REPORT NO. 300, "A BNF LIKE LANGUAGE FOR THE DESCRIPTION OF SYNTAX DIRECTED COMPILERS", BY H. R. G. TROUT. TBNF IS THE INPUT LANGUAGE FOR ST65, A COMPILER-COMPILER ON THE B6500 SYSTEM CURRENTLY IN CSL.
THIS FILE IS A DIRECT TRANSLATION OF THE SYNTAX OF OL/2 AS WRITTEN IN TBNF FOR TACOS, EXCEPT WHERE DIFFERENCES IN THE CAHARACTER SET NECESSITATED CHANGES.

```
<OL2PROGRAM> ::= LIST <STATEMENT>  ;

<STATEMENT> ::= <PL1STATEMENT> / <OL2STATEMENT>  ;
<PL1STATEMENT> ::= ## [<ANY> BUT ## ]* ## ;
<OL2STATEMENT> ::= <BLOCKLABELS>? [<NEWOL2BLOCK>/<PARTITIONSTATEMENT>/
          <SETSTATEMENT>/<INTERCHANGESTATEMENT>/<ENDSTATEMENT>/<OL2IFS>/
          <OL2IOSTATEMENT>/<OL2FORSTATEMENT>/<OL2ITERATIVECLAUSE>/
          <DEBUGSTATEMENT>/<OL2PROCEDURESTATEMENT>/<ASSIGNMENTSTATEMENT>]
          ;

<BLOCKLABELS> ::= [ <*I> : ]*  ;



<NEWOL2BLOCK> ::= LIST <IDENTIFIERDECLARATION> SEPARATOR <AND>  #;  ;

<IDENTIFIERDECLARATION> ::= <KEYWORD> <IDLIST> <DENOTATIVEPHRASE>
          <DEFINITION> <OTHERATTRIBUTES>?  ;

<KEYWORD> ::= LET / DEFINE / DENOTE BY  ;

<IDLIST> ::= LIST <IDENTIFIER> SEPARATOR <AND>  ;

<DENOTATIVEPHRASE> ::= TO? [BE / DENOTE ]? [ AN / A / THE ]?  ;

<DEFINITION> ::= <REALORCOMPLEX>? [ FINITE DIMENSIONAL ]? VECTOR
          [ SPACES / SPACE ] OF DIMENSION <NBYN> [ ,? [ WITH / WHICH
          [ HAS / CONTAINS ] AS ]? THE [ ELEMENTS / ELEMENT /
          MEMBERS / MEMBER ] [ A / THE ]? <SEQUENCE>? [ VECTORS /
          VECTOR ]? ]? / [<SEQUENCE>/<NBYN>/<LINEARITY>/<REALORCOMPLEX>]*
          <SECTION>? [<CLASS> [<SEQUENCE>/<MODULUS>]? <DIMENSION>?
          <BLOCKOF>? <ONORINASPACE>? <MODULUS>? / <BLOCKOF>?
          <ONORINASPACE>] / SCALARS / SCALAR  ;

<NBYN> ::= ( <BOUNDPAIREXPRESSION>  )  ;

<SEQUENCE> ::= [ WHICH [ ARE / IS A ]]? [ SEQUENCES / SEQUENCE ]
          OF? <MODULUS>? / <MODULUS>  ;

<MODULUS> ::= OF? [ MODULUS / MOD ] ( LIST <*N> SEPARATOR , ) OF?  ;

<LINEARITY> ::= [ <*N> / BI / TRI ]? -? LINEAR  ; %CONCATENATION ALLOWED HERE

<OTHERATTRIBUTES> ::= ( LIST [ <REALORCOMPLEX> / <FLOATORFIXED> /
          <BINARYORDECIMAL> / <PRECISION> ]  )  ;
```

```
<REALORCOMPLEX> ::= REAL / COMPLEX / CPLX  ;

<BINARYORDECIMAL> ::= BINARY / BIN / DECIMAL / DEC ;

<FLOATORFIXED> ::= FLOAT / FIXED  ;

<PRECISION> ::= ( LIST [[+/-]? <*N>] SEPARATOR ,  )  ;

<CLASS> ::= <IDENTITY>? [ ARRAYS / ARRAY / MATRIX / MATRICES /
          OPERATORS / #OPERATOR / OP ] <IDENTITY>? / [ VECTORS / VECTOR /
          VECS / VEC ]  ;

<IDENTITY> ::= IDENTITY / IDENTITIES / IDENT  ;

<DIMENSION> ::= [ WITH / OF ]? [ BOUNDS / BOUND / ORDER ]
          ( <BOUNDPAIREXPRESSION>  )  ;

<BLOCKOF> ::= [[[ UPPER / LOWER ] OFF? ]? DIAGONAL ]?
          <SEQUENCE> [ BLOCKS / BLOCK ] <SEQUENCE> OF <*I>  ;

<ONORINASPACE> ::= #ON <*I> / FROM (? LIST <SPACEID> SEPARATOR X )?
          [ INTO / TO / ONTO ] (? LIST <SPACEID> SEPARATOR X )? /
          [ ELEMENTS / ELEMENT / MEMBERS / MEMBER ] [ OF / IN] THE?
          VECTOR? SPACE? <*I>  ;

<SPACEID> ::= ( <*I> ) / <*I>  ;



<ENDSTATEMENT> ::= END <*I>? #;  ;



<ASSIGNMENTSTATEMENT> ::= LIST <OL2LEFTHANDSIDE> SEPARATOR , [ = / #<- ]
          <COMPAREEXPRESSION>  #;  ;

<OL2LEFTHANDSIDE> ::= <UNQUAL> / <REFFRENCE>  ;

<OL2ARITHMETICEXPRESSION> ::= LIST <OL2TERM> SEPARATOR [+/-]  ;

<OL2TERM> ::= LIST <OL2DIVIDE> SEPARATOR #*  ;

<OL2DIVIDE> ::= <OL2FACTOR> [ #/ [<MODIFIEDEXPRESSIONUNIT> /
          <EXTENDEDSCALAREXPRESSION> / <MODIFIEDOL2IDENTIFIER> ]]*  ;

<OL2FACTOR> ::= <OL2PRIMARY> [ #*#* <OL2EXTRA> ]?  ;

<OL2EXTRA> ::= [ <MODIFIEDEXPRESSIONUNIT> / <EXTENDEDSCALAREXPRESSION> ]
          [ #*#* <OL2EXTRA> ]?  ;

<OL2PRIMARY> ::= <MODIFIEDEXPRESSIONUNIT> / <EXTENDEDSCALAREXPRESSION> /
          <MODIFIEDOL2IDENTIFIER>  ;

<MODIFIEDEXPRESSIONUNIT> ::= [+/-]? ( <OL2ARITHMETICEXPRESSION> ) '?  ;

<MODIFIEDOL2IDENTIFIER> ::= [+/-]? <OL2IDENTIFIER>
          [ #[ <PL1EXPRESSION> #] ]? [ LIST [ #< LIST <PL1EXPRESSION>
```

```
        SEPARATOR , #> ]]? '?  ;

<OL2IUENTIFIER> ::= <*I>  ;

<EXTENUEUSCALAREXPRESSION> ::= [+/-]? <BASICS>  ;

<BASICS> ::= ( [+/-]? <BASICS> ) / <INNERPRODUCT> / <NORM> /
        <REFERENCE> / <CONSTANT>  ;

<REFERENCE> ::= LIST <BASICREF> SEPARATOR [-#>]  ;

<BASICREF> ::= LIST <UNQUAL> SEPARATOR .  ;

<UNQUAL> ::= <*I> [ #[ <PL1EXPRESSION> #] ]?  '?
        [ LIST [ #< LIST <PL1EXPRESSION> SEPARATOR , #> ] ]?
        [ ( LIST <OL2ARITHMETICEXPRESSION> SEPARATOR , ) ]?  ;

<CONSIANT> ::= [ <*R> / <*N> ] [ E [+/-] <*N> I? ]?  ;

<NORM> ::= || <OL2ARITHMETICEXPRESSION> ||  ;

<INNEKPRODUCT> ::= ( <UL2ARITHMETICEXPRESSION>  ,
        <OL2ARITHMETICEXPRESSIUN>  )  ;



<INTEKCHANGESTATEMENT> ::= INTERCHANGE <ROWORCOLUMN> <PL1EXPRESSION>
        AND <PL1EXPRESSION> [ IN / #ON / OF ] <ROOTPARTSEQ>  #;  ;



<PARTITIONSTATEMENT> ::= PARTITION <ROOTLIST> AFTER <ROWORCOLUMN>
        LIST <PL1EXPRESSION> SEPARATOR [ , / AND ] [ AND AFTER?
        <ROWORCOLUMN> LIST <PL1EXPRESSION> SEPARATOR [ , / AND ]]? #; ;

<ROWOKCOLUMN> ::= ROWS / ROW / COLUMNS / COLUMN / COLS / COL  ;

<ROOTLIST> ::= LIST <ROOTPARTSEQ> SEPARATOR <AND>  ;



<SETSIATEMENT> ::= SET LIST <SIMPLESETSTMT>
        SEPARATOR [ <AND> SET? ]  #;  ;

<SIMPLESETSTMT> ::= <IUENTIFIER> [[ EQUAL? TO? THE <SECTION> PART OF
        <ROOTPARTSEQ> ] / [ EQUAL / = / TO ] <TYPEDIDENT> ] ;

<TYPEUIDENT> ::= <ROOTPARTSEQ> <SECTION>? [ SCALAR / [ ROW / COLUMN /
        COL ] [ VECTOR / VEC ]? / MATRIX ]?  ;

<ROOTPARTSEQ> ::= <*I> [ #[ <PL1EXPRESSION> #] ]?
        [ #< LIST <PL1EXPRESSION> SEPARATOR , #> ]*  ;

<IDENIIFIER> ::= <*I> [ #[ <PL1EXPRESSION> #] ]?  ;

<SECTION> ::= BLOCK? TRI? [ DIAGONAL / DIAG ] / [[ STRICTLY /
        S. / S ]? [[ UPPER / LOWER ]? [ TRIANGULAR / TRIANG ] /
        [ UT / U.T. / LT / L.T. ]]] / [ SYMMETRIC / SYM ] /
        [ SELF [ ADJOINT / ADJ ] / [ RECTANGULAR / REC / SQUARE ]]  ;


<DEBUGSTATEMENT> ::= [ PRINT ID TABLE / PRINT TREE NODES /
        NODE PRINT OFF / TRACE #ON / TRACE OFF ]  #;  ;


<OL2PROCEDURESTATEMENT> ::= [ MAIN / RECURSIVE / REENTRANT ]*
        [ PROCEDURE / PROC ] [ ( LIST <ARGUMENT> SEPARATOR , ) ]? #;  ;

<ARGUMENT> ::= #*? <*I>  ;


<OL2IFS> ::= IF <OL2BOOLEANEXP> THEN <STATEMENT> [ ELSE <STATEMENT> ]? ;

<OL2BOOLEANEXP> ::= LIST <BOOLEANTERM> SEPARATOR |  ;

<BOOLEANTERM> ::= LIST <BOOLEANFACTOR> SEPARATOR #&  ;

<BOOLEANFACTOR> ::= ( <OL2BOOLEANEXP> ) / ¬<BOOLEANFACTOR> /
        <COMPAREEXPRESSION> [ <COMPAREOP> <COMPAREEXPRESSION> ]?  ;

<COMPAREOP> ::= #>= / #<= / ¬? [ = / #< / #> ]  ;

<COMPAREEXPRESSION> ::= #@ / NULL / <OL2ARITHMETICEXPRESSION>  ;


<OL2IOSTATEMENT> ::= [ INPUT / OUTPUT ] <LISTOFOPS>  #;  ;

<LISTOFOPS> ::= LIST <IOID> SEPARATOR <AND>  ;

<IOID> ::= <*I>  ;


<OL2FORSTATEMENT> ::= FOR <STEPPEDVARIABLE> = <SPECIFICATION>
        <CLAUSE>?  #;  ;

<STEPPEDVARIABLE> ::= <*I>  ;

<SPECIFICATION> ::= <UNIT> , LIST <UNIT> SEPARATOR , ,? .. .? <UNIT>  ;

<UNIT> ::= <PL1EXPRESSION>  ;

<CLAUSE> ::= [ WHILE / OR? UNTIL ] <OL2BOOLEANEXP>  ;


<OL2ITERATIVECLAUSE> ::= <CLAUSE> #;  ;
```

```
<BOUNDPAIREXPRESSION> ::= LIST [ <*N> / ( [+/-]? <*N> : [+/-]? <*N> ) ]
          SEPARATOR #;  ;

<AND> ::= , AND? / AND ;


<PL1EXPRESSION> ::= LIST <PL1TERM> SEPARATOR [ + / - ]  ;

<PL1TERM> ::= LIST <PL1FACTOR> SEPARATOR [ #* / #/ ]  ;

<PL1FACTOR> ::= <PL1BASIC> [ #*#* <PL1TERM> ]? / [+/-] <PL1TERM>  ;

<PL1BASIC> ::= ( <PL1EXPRESSION> ) / <PL1REFERENCE> / <PL1CONSTANT>  ;

<PL1REFERENCE> ::= LIST <PL1BASICREF> SEPARATOR [-#>]  ;

<PL1BASICREF> ::= LIST <PL1UNQUAL> SEPARATOR .  ;

<PL1UNQUAL> ::= <*I> [ ( LIST [ <PL1EXPRESSION> / #* ]
          SEPARATOR , ) ]?  ;

<PL1CONSTANT> ::= [ <*R> / <*N> ] [ E [+/-] <*N> ]?  ;

  END.
```

APPENDIX B

DOCUMENTATION OF OL2/TWST

The TWST65 compiler-compiler available on the B6500 uses two input files, known internally to TWST65 as SKELETON and CARD. The SKELETON file contains the basic routines for the compiler to be generated, including I/O and string, stack, and table manipulation. The CARD file contains the syntactic definitions and semantic routines which define the language involved. TWST65 produces an ALGOL source file called LANGUAGENAME/SOURCE by concatenating procedures to parse the language developed from the file CARD and the procedures contained in the file SKELETON. This ALGOL file is then compiled, yielding the final program.

OL2/TWST is the file equated to the CARD file in the generation of OL2/PARSER. Since OL/2TWST gives 'OL2' as the language name in the second line, TWST65 produces the ALGOL source file OL2/SOURCE, which is compiled as OL2/PARSER (Appendix D). OL2/TWST is written in TBNF (1, 2) to define a parsing algorithm which will accept correct statements as defined in OL2/SYNTAX (see Appendix A) and describe errors found in incorrect statements. The purpose of this appendix is to explain the specific constructs used in OL2/TWST, especially to those who may wish to modify OL2/PARSER in the future.

For those who are not familiar with TBNF, Appendix A contains a brief description of that form of BNF. The "TWST User's Manual" (2) is the best source of information regarding how a parsing algorithm produced by TWST65 will be affected by the use of various features in TBNF.

TWST65 is normally used to generate compilers; however, it served very well in generating OL2/PARSER. There are no semantic routines in the usual sense in OL2/TWST, for OL2/PARSER is concerned solely with the recognition of proper OL/2 statements. There is no definition of <PROGRAM> because an OL/2 program, syntactically, is only a list (to use TWST65 terminology) of OL/2 statements. Future modifications to OL2/PARSER could very well include semantic procedures to incorporate features as described in Section 4 of this thesis.

Errors and Other Messages

Those sections of OL2/TWST which have the form of semantic routines are almost exclusively concerned with displaying appropriate error messages. There are approximately one hundred and fifty separate error messages and another dozen messages identifying statement types. They are all encoded as semantic routines which write the appropriate format to a file named STATION.

For example,

```
        <E0101> ::= @S[USEPOINT; WRITE (STATION, F0101);];
```

causes the message in F0101,

```
        <NEWOL2BLOCK>: TERMINAL ';' MISSING ,
```

to be displayed. USEPOINT is a procedure contained in OL2/SKELETON (Appendix C) which displays an error marker under the first character not scanned when an error is found.

All of these message-producing routines are defined as nonterminals; the error routines are all alternatives within the syntax. Consequently, they are treated exactly as the syntactic nonterminals are treated, except that they are always satisfied unless action is taken to cause the parse to fail. It is desirable to cause the parse to fail to prevent one error from precipitating a large number of subsequent non-informative error messages.

Consider, for example, what would happen if the following string were parsed as a <PL1EXPRESSION> if errors did not cause the parse to fail.

```
A - ( B*/C)
```

The routine would expect a <PL1FACTOR> following the "*". The "/" does not parse as a <PL1FACTOR>, but an error routine (<E1302>) would satisfy the parse. Since the parse did not fail 'C' would be examined, and because 'C' is not an operator symbol, the parse would expect a closing parenthesis. <E1306> would satisfy the parse at this point and the ')' would be examined. The parse would continue spewing forth error messages until it ran out of input. Therefore, all error routines except those concerning terminal semicolons are of the form

```
<Exxxx> ::= @T[USEPOINT; WRITE(STATION, Fxxxx);]; .
```

When TWST65 inserts the ALGOL code from an '@T' semantic routine into OL2/SOURCE, it provides code to cause the alternative containing that semantic routine to fail if a Boolean variable called SEMANTICTEST is FALSE at the conclusion of the routine. USEPOINT always sets SEMANTICTEST to the value FALSE. Consequently, most of the error routines cause the parsing algorithm to examine the next alternative in the syntax.

The effect of the above is to produce a series of error messages related to the first error found. The error messages begin with as specific a message as possible and continue until the most general message concerning the error is displayed. The effect is to indicate the path taken by the algorithm through the parsing tree to the point of the error. After the algorithm backtracks to produce the most general error, all subsequent alternatives are preceded by a semantic routine of the form

```
@T[ SEMANTICTEST := USEPT ; ] ; ,
```

which prevent the algorithm from examining any following alternatives. Eventually, the parse will fail without attempting to locate any further errors.

Semantic routines of the form

```
@T[ SEMANTICTEST := USEPT ; ] ;
```

prevent the algorithm from examining the alternatives they precede since USEPT is set to the value FALSE by USEPOINT when an error is found and is not set to the value TRUE until the next line of input is accepted. This semantic routine is used throughout OL2/TWST for the stated purpose. For example, in the definition of <OL2STATEMENT>, each of the last eleven alternative statement types are preceded by this semantic routine. Thus, if an error is found in a <PARTITIONSTATEMENT>, the last ten alternative statement types are never examined.

Other Constructs in OL2/TWST

The remainder of this discussion will cover various syntactic definitions in OL2/TWST in the order in which they occur in the listing. The definitions chosen will illustrate features that are not straightforward; they usually offer insight into the workings of TWST65.

<FINISH>

<FINISH> is included as the first alternative in <STATEMENT> so that the entry of the word 'FINISH' at the terminal will cause OL2/PARSER to terminate as quickly as possible.

<BLOCKLABELS>

Labels may occur before any statement. When an identifier string followed by a colon is found, the semantic routine sets bit 46 of the first word in the table entry created by TWST65's table building routine to indicate that this identifier is a label. The table maintained by standard TWST65 routines, contained in OL2/SKELETON, and the use of the variable 'P1' is discussed in Trout (1, 2).

This method does not allow for catching duplicate labels, although it could be expanded to insure that the same identifier is not used previously as a variable (see <NEWID>).

<IDENTIFIERDECLARATION>

<PARSE01> is placed immediately following <KEYWORD> so that the user will learn as soon as possible when OL2 PARSER considers his entry to be a <NEWOL2BLOCK>. The same strategy is used for each statement type.

<LINEARITY>

All possible combinations are spelled out since the string handling procedures used in OL2/PARSER expect defined words to be delimited by non-alphanumeric characters. They will not recognize 'BILINEAR' as 'BI' followed by 'LINEAR'.

<ONORINASPACE>

The character 'X' is used instead of 'x', mathematical multiplication operator, because of character set limitations. Since 'X' is alphanumeric, 'AXB' is not equivalent to 'A X B'.

The use of optional parentheses around LIST's of <SPACEID>, and the possibility of parentheses around the <*I> in <SPACEID> prevents OL2/PARSER from recognizing all correct constructions, since the first parenthesis found is always assumed to be the optional left parenthesis. Thus,

```
FROM (A) X B INTO C
```

will not be accepted, although syntactically correct. The right parenthesis is parsed as the optional right parenthesis, so the parse expects 'INTO' or 'ONTO' or 'TO' when it reaches 'X'.

<AND>

This nonterminal is introduced to reduce the amount of typing, although it does introduce one complication (see <PARTITIONSTATEMENT>).

<NEWID>

This nonterminal contains the semantic routine to mark the identifier table for correctly declared new identifiers. Future modifications to this routine should probably include marking different bits for different types of identifiers. The available bits accessible through 'TABLE[P1]' (see (2)) are the six high order (42-47) bits of the initial table entry.

<CHECKID>

This nonterminal contains the semantic routine which checks to see whether a given identifier has been previously declared in a <NEWOL2BLOCK>, <SETSTATEMENT>, or <OL2FORSTATEMENT>, in which <NEWID> is used. Future modifications should probably include testing routines to see if identifier usage is consistent with identifier declaration (discussed above).

<PARTITIONSTATEMENT>

', AND? / AND' is used instead of <AND> in this production because, in a statement which partitions after both rows and columns, the single word 'AND' must separate the row and column designations. If <AND> were used in the previous SEPARATOR construct, the word "AND" would always be reduced to <AND>, and the word would not be recognized, causing the parse to fail whenever a double (row and column) designation is used.

The series of 'NOT terminal''s serves to condition the SEPARATOR so that the word 'AND' separating row and c umn designations will not be reduced as a separator in the preceding list.

<OL2FORSTATEMENT>

The integer variable ALTERNATIVE is given the value '2' before scanning for <CLAUSE> to suppress the message of <PARSE09> if <CLAUSE> is indeed found.

<OL2ITERATIVECLAUSE>

Since this nonterminal is nothing but a <CLAUSE> followed by a semicolon, there is no use in constructing an equivalent definition and set of error messages. ALTERNATIVE is given the value '1' before scanning for <CLAUSE> so that <PARSE09> will display the appropriate statement type message.

<OL2BOOLEANEXP>

```
        <BOOLEANTERM>[| <BOOLEANTERM>]*
```

is used instead of

```
        LIST <BOOLEANTERM> SEPARATOR |
```

because the second form would not force the algorithm to find a <BOOLEANTERM> following a '|' character or produce an error. The second form would lead the parse to identify the '|' as an incorrect syntactic unit following a correct <OL2BOOLEANEXP>, since the LIST-SEPARATOR construct does not consider an occurrence of a separator element to be a separator until the following list element is successfully found.

The first form at least will yield a message placing the error after the previous '|'.

A similar approach to errors following separating symbols is used in <REFERENCE>, <PL1REFERENCE>, <OL2ARITHMETIC-EXPRESSION> , and <PL1EXPRESSION>.

<UNQUAL>

'#< NOT - NOT = ' is used in the third line so that '<-', a replacement operator in <ASSIGNMENTSTATEMENT>, and '<=', a comparing operator in <OL2BOOLEANEXP>, cannot be considered the '<' at the beginning of a subarray designation.

RESERVE

Usually TWST65 reserves all defined words (alphanumeric strings beginning with an alphabetic character) found as terminals in the syntax for a language. Reserved words will cause the identifier scanning atom, *I , to fail. OL2/TWST contains such defined words as 'A', 'I', 'E', and 'X', which are commonly used as variable names.

Inserting a line 'RESERVE ; ' just before the 'END.' line in OL2/TWST would prevent TWST65 from reserving any of the defined words found in the listing. If this were done, however, the keywords which are specific to the various statement types would be recognized as identifiers during the scan for <BLOCKLABELS>, and would not be recognized to direct the parsing algorithm.

Inserting a line 'RESERVE word list', as in OL2/TWST, causes only the words in the list to be reserved. The word 'END' cannot appear in this list because of a bug ir TWST65, but another bug in TWST65 causes the first word following the reserved words in the identifier table built by TWST65 to be reserved also. Therefore, <ENDSTATEMENT> is placed before <PL1EXPRESSION> so that 'END' is the last word defined and the first word after the reserved words.

REMARKS ON ALTERNATIVES

It is often important to consider the order in which alternatives are listed, for once the parsing algorithm reduces a portion of an input string to a particular nonterminal, that portion of the string is never again examined. Backtracking can succeed only if subsequent alternatives specify the same nonterminals in the same order over that portion of the string.

For example, if a production contains the alternatives

```
          ( <OL2ARITHMETICEXPRESSION> )
          /
          ( <PL1EXPRESSION> , <PL1EXPRESSION> )
```

then the string "(A*B , C/D)" will not be parsed as the second alternative since "A*B" reduces to <OL2ARITHMETICEXPRESSION> before the "," fails to parse as a ")".

INSERTION OF SEMANTIC ROUTINES

Approximately half-way through the listing of OL2/TWST, which follows this discussion, an 'END.' line separates the syntax and inline semantic routines from the format declarations for OL2/PARSER. TWST65 uses everything preceding the 'END.' line in generating the parsing algorithm of OL2/PARSER. Anything following the 'END.' line is inserted into the ALGOL code file OL2/SOURCE produced by TWST65. Large or often used semantic routines may be inserted in this manner; for example, USEPOINT (discussed in Appendix C) could have been placed after the 'END.' line rather than in OL2/SKELETON.

Large semantic routines must be handled in this manner, since TWST65 enforces a limit on the length of inline semantic routines. If the TWST65 generated variables FIRST, LAST, or any of the related PMx or Pxx variables (2) are needed, then the semantic procedure must be called from within a short inline "@S" or "@T" semantic routine.

Also, more efficient procedure linkage will result if semantic procedures are placed after the 'END.' line than by defining them as nonterminals within the syntax, since TWST65 will not set up all the tests associated with nonterminals.

FILE LISTING

The file OL2/TWST is maintained as shown in Appendix D; a listing follows this page.

```
$ CARD LIST SYNTAX
  OL2
<STATEMENT> ::= <FINISH> / <PL1STATEMENT> / <OL2STATEMENT>  ;
<FINISH> ::= FINISH  @W[ FINISH := TRUE ; ] ;
<PL1STATEMENT> ::= ## [<ANY> BUT ## ]* ## <E1300>  ;
<OL2STATEMENT> ::= <BLOCKLABELS> [ <NEWOL2BLOCK> /
          @T[ SEMANTICTEST := USEPT ; ] <PARTITIONSTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <SETSTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <INTERCHANGESTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <ENDSTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <OL2IFS> /
          @T[ SEMANTICTEST := USEPT ; ] <OL2IOSTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <OL2FORSTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <OL2ITERATIVECLAUSE> /
          @T[ SEMANTICTEST := USEPT ; ] <DEBUGSTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <OL2PROCEDURESTATEMENT> /
          @T[ SEMANTICTEST := USEPT ; ] <ASSIGNMENTSTATEMENT> ]  ;
<BLOCKLABELS> ::= [ <*I> : @S[ TABLE[P1].[46:1] := 1 ; ] ]*  ;

<NEWOL2BLOCK> ::= LIST <IDENTIFIERDECLARATION> SEPARATOR <AND>
          [ #; / <E0101> ] ;
<IDENTIFIERDECLARATION> ::= <KEYWORD> <PARSE01> [ <IDLIST> / <E0102> ]
          <DENOTATIVEPHRASE>? [ <DEFINITION> / <E0103> ]
          <OTHERATTRIBUTES>?  ;
<KEYWORD> ::= LET / DEFINE / DENOTE BY  ;
<IDLIST> ::= LIST [ <NEWIDENTIFIER> / <E0104> ] SEPARATOR <AND> ;
<DENOTATIVEPHRASE> ::= TO? [ BE / DENOTE ]  [ AN / A / THE ]?  ;
<DEFINITION> ::= <REALORCOMPLEX>? [ FINITE [ DIMENSIONAL / <E0105> ]]?
          VECTOR [ SPACES / SPACE / <E0106> ] [ OF / <E0107> ]
          [ DIMENSION / <E0108> ] [ <NBYN> / <E0109> ]
          [ ,? [ WITH / WHICH [ HAS / CONTAINS / <E0110> ]
          [ AS / <E0111> ]]? THE [ ELEMENTS / ELEMENT / MEMBERS / MEMBER
          / <E0112> ] [ A / THE ]? <SEQUENCE>? [ VECTORS / VECTOR ]? ]?
          [ <SEQUENCE> / <NBYN> / <LINEARITY> / <REALORCOMPLEX> ]*
          / @T[ SEMANTICTEST := USEPT ; ]
          <SECTION>? [ <CLASS> [ <SEQUENCE> / <MODULUS> ]? <DIMENSION>?
          <BLOCKOF>? <ONORINASPACE>? <MODULUS>? / <BLOCKOF>?
          <ONORINASPACE> ]
          / @T[ SEMANTICTEST := USEPT ; ]
          SCALARS / SCALAR  ;
<NBYN> ::= ( [ <BOUNDPAIREXPRESSION> / <E0113> ] [ ) / <E0114> ]  ;
<SEQUENCE> ::= [ WHICH [ ARE / IS A / <E0115> ]]? [ SEQUENCES /
           SEQUENCE ] OF? <MODULUS>? /@T[SEMANTICTEST:=USEPT;] <MODULUS>;
<MODULUS> ::= OF? [ MODULUS / MOD ] [ ( / <E0116> ]
           LIST [ <PL1EXPRESSION> / <E0117> ] SEPARATOR ,
          [ ) / <E0119> ] OF?  ;
<LINEARITY> ::= LINEAR / BILINEAR / TRILINEAR / <*N>LINEAR /
          <*N>-LINEAR / BI-LINEAR / TRI-LINEAR  ;
<OTHERATTRIBUTES> ::= ( LIST [ <ATTRIBUTE> / <E0120> ] [ ) / <E0121> ] ;
<ATTRIBUTE> ::= <REALORCOMPLEX> / <BINARYORDECIMAL> / <FLOATORFIXED> /
          <PRECISION>  ;
<CLASS> ::= <IDENTITY>? [ ARRAYS / ARRAY / MATRICES / MATRIX /
          OPERATORS / OPERATOR / OP ] <IDENTITY>?
          / [ VECTORS / VECTOR / VECS / VEC ]  ;
<IDENTITY> ::= IDENTITY / IDENTITIES / IDENT  ;
<REALORCOMPLEX> ::= REAL / COMPLEX / CPLX  ;
<BINARYORDECIMAL> ::= BINARY / BIN / DECIMAL / DEC  ;
```

```
<FLOATORFIXED> ::= FLOAT / FIXED  ;
<PRECISION> ::= ( [ + / - ]? [ <*N> / <E0122> ] [ , [ + / - ]?
        [ <*N> / <E0123> ] ]? [ ) / <E0124> ]  ;
<DIMENSION> ::= [ WITH / OF ]? [ BOUNDS / BOUND / ORDER ]
        [ <NBYN> / <E0125> ]  ;
<BLOCKOF> ::=  [[[ UPPER / LOWER ] OFF? ]? DIAGONAL ]? <SEQUENCE>
        [ BLOCKS / BLUCK / <E0128> ] [ <SEQUENCE> /<E0129> ]
        [ OF / <E0130> ] [ <CHECKID> / <E0131> ]  ;
<ONORINASPACE> ::= #ON [ <CHECKID> / <E0132> ]
        / @T[ SEMANTICTEST := USEPT ; ]
        FROM (? LIST [ <SPACEID> / <E0133> ] SEPARATOR X
        )? [ INTO / ONTO / TO / <E0135> ]
        (? LIST [ <SPACEID> / <E0136> ] SEPARATUR X )?
        / @T[ SEMANTICTEST := USEPT ; ]
        [ ELEMENTS / ELEMENT / MEMBERS / MEMRER ] [ OF / IN / <E0138> ]
        THE? VECTOR? SPACE? [ <*I> / <E0139> ]  ;
<AND> ::= , AND? / AND  ;
<SPACEID> ::= ( [ <CHECKID> / <E0140> ] [ ) / <E0141> ] / <CHECKID> /
        <E0142>  ;
<IDENTIFIER> ::= [ <CHECKID> / <E0316> ]
        [ #[ [ <PL1EXPRESSION> / <E0317> ] [ #] / <E0318> ]]?  ;
<NEWIDENTIFIER> ::= [ <NEWID> / <E0316> ]
        [ #[ [ <PL1EXPRESSION> / <E0317> ] [ #] / <E0318> ] ]?  ;
<NEWID> ::= <*I> @S[ TABLE[P1].[47:1] := 1 ; ] ;
<CHECKID> ::= <*I>
     @S[                IF TABLE[P1].[47:1] = 0 THEN BEGIN
                   TABLE[P1].[47:1] := 1  ;
                   PT := POINTER(IDNAME[0]) + 10  ;
                   REPLACE PT BY " " FOR 26 ;
                   IF TABLE[P1].COUNTFIELD LEQ 26 THEN
                   REPLACE PT BY POINTER(TABLE[P1 + 1])
                            FOR TABLE[P1].COUNTFIELD
                   ELSE REPLACE PT BY POINTER(TABLE[P1 + 1]) FOR 26 ;
                   WRITE (STATION, 12, IDNAME[*]) ;  END ; ] ;

<PARTITIONSTATEMENT> ::= PARTITION <PARSE02> [ <ROOTLIST> / <E0201> ]
        [ AFTER / <E0202> ] [ <ROWORCOLUMN> / <E0203> ]
        LIST [ <PL1EXPRESSION> / <E0204> ] SEPARATOR [ [ , AND? / AND ]
        NOT AFTER NOT ROW NOT ROWS NOT COLUMN NOT COLUMNS NOT COLS
        NOT COL ] [ AND AFTER? [ <ROWORCOLUMN> / <E0206> ]
        LIST [ <PL1EXPRESSION> / <E0207> ] SEPARATUR <AND> ]?
        [ #; / <E0209> ]  ;
<ROWORCOLUMN> ::= ROWS / ROW / COLUMNS / COLUMN / COLS / COL  ;
<ROOTLIST> ::= LIST [ <ROOTPARTSEQ> / <E0210> ] SEPARATUR <AND>  ;

<SETSTATEMENT> ::= SET <PARSE03> LIST [ <SIMPLESETSTMT> / <E0301> ]
        SEPARATOR [ <AND> SET? ] [ #; / <E0303> ]      ;
<SIMPLESETSTMT> ::= [ <NEWIDENTIFIER> / <E0304> ]
        [ [ EQUAL? TO? THE [ <SECTION> / <E0305> ] [ PART / <E0306> ]
        [ OF / <E0307> ] [ <ROOTPARTSEQ> / <E0308> ]]
        / @T[ SEMANTICTEST := USEPT ; ]
        [ EQUAL / = / TO ] [ <TYPEDIDENT> / <E0309> ] ]  ;
<TYPEDIDENT> ::= [ <ROOTPARTSEQ> / <E0310> ] <SECTION>?
        [ SCALAR / [ ROW / COLUMN / COL ] [ VECTOR / VEC ]? /
        MATRIX ]?  ;
<ROOTPARTSEQ> ::= <CHECKID>
        [ #[ [ <PL1EXPRESSION> / <E0312> ] [ #] / <E0313> ] ]?
        [ [ #< NOT = ] LIST [ <PL1EXPRESSION> / <E0314> ] SEPARATOR .
        [ #> / <E0315> ]]*  ;
<SECTION> ::= BLOCK? [ DIAGONAL / TRIDIAGONAL / DIAG / TRIDIAG ] /
        [[ STRICTLY / S. / S ]? [[ UPPER / LOWER ]?
        [ TRIANGULAR / TRIANG ] / [ UT / U.T. / LT / L.T. ]]]
        [ SYMMETRIC / SYM ] /
        SELF [ ADJOINT / ADJ / <E0319> ] /
        [ RECTANGULAR / REC / SQUARE ]  ;

<INTERCHANGESTATEMENT> ::= INTERCHANGE <PARSE04> [ <ROWORCOLUMN> /
        <E0401> ] [ <PL1EXPRESSION> / <E0402> ] [ AND / <E0403> ]
        [ <PL1EXPRESSION> / <E0404> ] [ IN / OF / #ON / <E0405> ]
        [ <ROOTPARTSEQ> / <E0406> ] [ #; / <E0407> ] ;

<OL2IFS> ::= IF <PARSE06> [ <OL2BOOLEANEXP> / <E0601> ] [ THEN /<E0602>]
        [ <STATEMENT> / <E0603> ] [ ELSE [ <STATEMENT> / <E0604> ]]? ;

<OL2IOSTATEMENT> ::= [ INPUT / OUTPUT ] <PARSE07>
        [ <LISTOFOPS> / <E0701> ] [ #; / <E0702> ]  ;
<LISTOFOPS> ::= LIST [ <IOID> / <E0703> ] SEPARATOR , ;
<IOID> ::= <CHECKID>  ;

<OL2FORSTATEMENT> ::= FOR <PARSE08> [ <STEPPEDVARIABLE> / <E0801> ]
        [ = / <E0802> ] [ <SPECIFICATION> / <E0803> ]
       @Q[ ALTERNATIVE := 2 ; ] <CLAUSE>? [ #; / <E0804> ]  ;
<STEPPEDVARIABLE> ::= <NEWID>  ;
<SPECIFICATION> ::= [ <UNIT> / <E0805> ] [ , / <E0806> ]
        [ <UNIT> / <E0807> ] [ [ , NOT . ] [ <UNIT> / <E0807> ] ]* ,?
        [ .. / <E0808> ] .* ,? [ <UNIT> / <E0809> ]  ;
<UNIT> ::= <PL1EXPRESSION>  ;

<OL2ITERATIVECLAUSE> ::= @Q[ ALTERNATIVE := 1 ; ] <CLAUSE>
        [ #; / <E0901> ]  ;
<CLAUSE> ::= [ WHILE / OR? UNTIL ] <PARSE09>
        [ <OL2BOOLEANEXP> / <E0902> ]  ;
<OL2BOOLEANEXP> ::= [ <BOOLEANTERM> / <E0903> ]
        [ [ | NOT | ] [ <BOOLEANTERM> / <E0910> ] ]*  ;
<BOOLEANTERM> ::= [ <BOOLEANFACTOR> / <E0904> ]
        [ #& [ <BOOLEANFACTOR> / <E0911> ] ]*  ;
<BOOLEANFACTOR> ::= ( [ <OL2BOOLEANEXP> / <E0905> ] [ ) / <E0906> ] /
        @T[ SEMANTICTEST := USEPT ; ] ¬ [ <BOOLEANFACTOR> / <E0907> ] /
        @T[ SEMANTICTEST := USEPT ; ] [ <COMPAREEXPRESSION> / <E0908> ]
        [ <COMPAREOP> [ <COMPAREEXPRESSION> / <E0909> ] ]?  ;
<COMPAREOP> ::= #>= / #<= / ¬? [ = / #> / #< ]  ;
<COMPAREEXPRESSION> ::= #@ / NULL / <OL2ARITHMETICEXPRESSION>  ;

<DEBUGSTATEMENT> ::= [ PRINT <PARSE10> [ ID [ TABLE / <E1001> ] /
                              [ TREE / <E1002> ] [ NODES / <E1003> ]] /
            NODE <PARSE10> [ PRINT / <E1004> ] [ OFF / <E1005> ] /
            TRACE <PARSE10> [ #ON / OFF / <E1006> ]] [ #; / <E1007> ] ;

<OL2PROCEDURESTATEMENT> ::= [ MAIN / RECURSIVE / REENTRANT ]*
        [ PROCEDURE / PROC ] <PARSE11>
        [ ( LIST [ <ARGUMENT> / <E1101> ] SEPARATOR , [ ) / <E1102> ]]?
        [ #; / <E1103> ]  ;
<ARGUMENT> ::= #*? [ <NEWID> ]  ;
```

```
<ASSIGNMENTSTATEMENT> ::= LIST [ <OL2LEFTHANDSIDE> / <E1201> ]
        SEPARATOR , [ = / #<- / <E1203> ]  <PARSE12>
        [ <COMPAREEXPRESSION> / <E1204> ] [ #; / <E1205> ]  ;
<OL2LEFTHANDSIDE> ::= @Q[ALTERNATIVE:=3;] [ <REFERENCE> / <UNQUAL> ]  ;
<REFERENCE> ::= <BASICREF> [ [ -#> ] [ <BASICREF> / <E1206> ] ]  ;
<BASICREF> ::= <UNQUAL> [ . [ <UNQUAL> / <E1207> ] ]*  ;
<UNQUAL> ::= <CHECKID>
        [ #[ [ <PL1EXPRESSION> / <E1211> ] [ #] / <E1212> ]]? '?
        [ [ #< NOT - NOT = ] LIST [ <PL1EXPRESSION> / <E1213> ]
        SEPARATOR , [ #> / <E1215> ] ]*
        [ [ ( @T[ IF ALTERNATIVE = 4 THEN SEMANTICTEST := FALSE ; ]]
        LIST [ <OL2ARITHMETICEXPRESSION> / <E1216> ] SEPARATOR,
                      [ ) / <E1218> ]]?  ;
<OL2ARITHMETICEXPRESSION> ::= <OL2TERM> [ [ + / - ] [ <OL2TERM> /
        <E1219> ] ]*  ;
<OL2TERM> ::= <OL2DIVIDE> [ #* [ <OL2DIVIDE> / <E1220> ] ]*  ;
<OL2DIVIDE> ::= <OL2FACTOR> [ #/ [ <OL2PRIMARY> / <E1222> ] ]*  ;
<OL2FACTOR> ::= [ <OL2PRIMARY> / <E1223> ]
        [ #*#* [ <OL2EXTRA> / <E1224> ]]?  ;
<OL2EXTRA> ::= [ <EXTENDEDSCALAREXPRESSION> / <MODIFIEDEXPRESSIONUNIT> ]
        [ #*#* [ <OL2EXTRA> / <E1226> ] ]?  ;
<OL2PRIMARY> ::= <MODIFIEDOL2IDENTIFIER> / <EXTENDEDSCALAREXPRESSION> /
        <MODIFIEDEXPRESSIONUNIT>  ;
<MODIFIEDEXPRESSIONUNIT> ::= [ + / - ]? (
        [ <OL2ARITHMETICEXPRESSION> / <E1227> ] [ ) / <E1228> ] '?  ;
<MODIFIEDOL2IDENTIFIER> ::= [ + / - ]? @Q[ ALTERNATIVE := 4 ; ]
        <REFERENCE>  '?  ;
<EXTENDEDSCALAREXPRESSION> ::= [ + / - ]? <BASICS>  ;
<BASICS> ::= <NORM> /
        @T[ SEMANTICTEST := USEPT ; ] <CONSTANT> /
        @T[ SEMANTICTEST := USEPT ; ] <INNERPRODUCT> /
        @T[ SEMANTICTEST := USEPT ; ] ( [ + / - ]? <BASICS>
                        [ ) / <E1234> ] /
        @T[ SEMANTICTEST := USEPT ; ALTERNATIVE := 3 ;] <REFERENCE>  ;
<NORM> ::= || [ <OL2ARITHMETICEXPRESSION> / <E1240> ]
        [ || / <E1241> ]  ;
<CONSTANT> ::= [ <*R> / <*N> ] [ E [ + / - / <E1242> ]
        [ <*N> / <E1243> ] ]? ]?  ;
<INNERPRODUCT> ::= ( [ <OL2ARITHMETICEXPRESSION> AHEAD , ]
        [ , / <E1236> ] [ <OL2ARITHMETICEXPRESSION> / <E1237> ]
        [ ) / <E1238> ]  ;
<BOUNDPAIREXPRESSION> ::= LIST [ <PL1EXPRESSION> / [ ( / <E1244> ]
        [ + / - ]? [ <PL1EXPRESSION> / <E1245> ] [ : / <E1246> ]
        [ + / - ]? [ <PL1EXPRESSION> / <E1247> ] [ ) / <E1248> ] ]
        SEPARATOR #;  ;

<ENDSTATEMENT> ::= END <PARSE05> <CHECKLABEL>? [ #; / <E0501> ] ;
<CHECKLABEL> ::= <*I>
        @S[        IF TABLE[P1].[46:1] = 0 THEN BEGIN
                   PT := POINTER(IDNAME[0]) + 10  ;
                   REPLACE PT BY " " FOR 26  ;
                   IF TABLE[P1].COUNTFIELD LEQ 26 THEN
                   REPLACE PT BY POINTER(TABLE[P1 + 1])
                        FOR TABLE[P1].COUNTFIELD
                   ELSE REPLACE PT BY POINTER(TABLE[P1 + 1]) FOR 26  ;
                   WRITE (STATION, 12, IDNAME[*])  ;  END  ;  ] ;
```

```
<PL1EXPRESSION> ::= <PL1TERM> [ [ + / - ] [ <PL1TERM> / <E1301> ] ]*  ;
<PL1TERM> ::= <PL1FACTOR> [ [ #* / #/ ] [ <PL1FACTOR> / <E1302> ] ]*  ;
<PL1FACTOR> ::= [ + / - ] [ <PL1FACTOR> / <E1303> ] /
        <PL1BASIC> [ #*#* [ <PL1FACTOR> / <E1304> ] ]?  ;
<PL1BASIC> ::= ( [ <PL1EXPRESSION> / <E1305> ] [ ) / <E1306> ]
        <PL1REFERENCE> / <PL1CONSTANT>  ;
<PL1REFERENCE> ::= <PL1BASICREF> [ [ -#> ] [ <PL1BASICREF> / <E1310>]]*;
<PL1BASICREF> ::= <PL1UNQUAL> [ . [ <PL1UNQUAL> / <E1311> ] ]*  ;
<PL1UNQUAL> ::= <CHECKID> [ ( LIST [ <PL1EXPRESSION> / #* /
        <E1307> ] SEPARATOR , [ ) / <E1308> ] ]?  ;
<PL1CONSTANT> ::= [ <*R> / <*N> ] [ E [ + / - ]?
        [ <*N> / <E1309> ] ]?  ;



<E0101> ::= @S[USEPOINT;WRITE(STATION,F0101);WRITE(STATION,FSEP);];
<E0102> ::= @T[USEPOINT;WRITE(STATION,F0102);];
<E0103> ::= @T[USEPOINT;WRITE(STATION,F0103);WRITE(STATION,FSEP);];
<E0104> ::= @T[USEPOINT;WRITE(STATION,F0104);];
<E0105> ::= @T[USEPOINT;WRITE(STATION,F0105);];
<E0106> ::= @T[USEPOINT;WRITE(STATION,F0106);];
<E0107> ::= @T[USEPOINT;WRITE(STATION,F0107);];
<E0108> ::= @T[USEPOINT;WRITE(STATION,F0108);];
<E0109> ::= @T[USEPOINT;WRITE(STATION,F0109);];
<E0110> ::= @T[USEPOINT;WRITE(STATION,F0110);];
<E0111> ::= @T[USEPOINT;WRITE(STATION,F0111);];
<E0112> ::= @T[USEPOINT;WRITE(STATION,F0112);];
<E0113> ::= @T[USEPOINT;WRITE(STATION,F0113);];
<E0114> ::= @T[USEPOINT;WRITE(STATION,F0114);WRITE(STATION,FSEP);];
<E0115> ::= @T[USEPOINT;WRITE(STATION,F0115);];
<E0116> ::= @T[USEPOINT;WRITE(STATION,F0116);];
<E0117> ::= @T[USEPOINT;WRITE(STATION,F0117);];
<E0119> ::= @T[USEPOINT;WRITE(STATION,F0119);WRITE(STATION,FSEP);];
<E0120> ::= @T[USEPOINT;WRITE(STATION,F0120);];
<E0121> ::= @T[USEPOINT;WRITE(STATION,F0121);];
<E0122> ::= @T[USEPOINT;WRITE(STATION,F0122);];
<E0123> ::= @T[USEPOINT;WRITE(STATION,F0123);];
<E0124> ::= @T[USEPOINT;WRITE(STATION,F0124);WRITE(STATION,FSEP);];
<E0125> ::= @T[USEPOINT;WRITE(STATION,F0125);];
<E0128> ::= @T[USEPOINT;WRITE(STATION,F0128);];
<E0129> ::= @T[USEPOINT;WRITE(STATION,F0129);];
<E0130> ::= @T[USEPOINT;WRITE(STATION,F0130);];
<E0131> ::= @T[USEPOINT;WRITE(STATION,F0131);];
<E0132> ::= @T[USEPOINT;WRITE(STATION,F0132);];
<E0133> ::= @T[USEPOINT;WRITE(STATION,F0133);];
<E0135> ::= @T[USEPOINT;WRITE(STATION,F0135);WRITE(STATION,FSEP);];
<E0136> ::= @T[USEPOINT;WRITE(STATION,F0136);];
<E0138> ::= @T[USEPOINT;WRITE(STATION,F0138);];
<E0139> ::= @T[USEPOINT;WRITE(STATION,F0139);];
<E0140> ::= @T[USEPOINT;WRITE(STATION,F0140);];
<E0141> ::= @T[USEPOINT;WRITE(STATION,F0141);];
<E0142> ::= @T[USEPOINT;WRITE(STATION,F0142);];
<E0201> ::= @T[USEPOINT;WRITE(STATION,F0201);];
<E0202> ::= @T[USEPOINT;WRITE(STATION,F0202);WRITE(STATION,FSEP);];
<E0203> ::= @T[USEPOINT;WRITE(STATION,F0203);];
<E0204> ::= @T[USEPOINT;WRITE(STATION,F0204);];
<E0206> ::= @T[USEPOINT;WRITE(STATION,F0206);];
```

```
<E0207> ::= @T[USEPOINT;WRITE(STATION,F0207);];
<E0209> ::= @S[USEPOINT;WRITE(STATION,F0209);WRITE(STATION,FSEP);];
<E0210> ::= @T[USEPOINT;WRITE(STATION,F0210);];
<E0301> ::= @T[USEPOINT;WRITE(STATION,F0301);];
<E0303> ::= @S[USEPOINT;WRITE(STATION,F0303);WRITE(STATION,FSEP);];
<E0304> ::= @T[USEPOINT;WRITE(STATION,F0304);];
<E0305> ::= @T[USEPOINT;WRITE(STATION,F0305);];
<E0306> ::= @T[USEPOINT;WRITE(STATION,F0306);];
<E0307> ::= @T[USEPOINT;WRITE(STATION,F0307);];
<E0308> ::= @T[USEPOINT;WRITE(STATION,F0308);];
<E0309> ::= @T[USEPOINT;WRITE(STATION,F0309);];
<E0310> ::= @T[USEPOINT;WRITE(STATION,F0310);];
<E0312> ::= @T[USEPOINT;WRITE(STATION,F0312);];
<E0313> ::= @T[USEPOINT;WRITE(STATION,F0313);];
<E0314> ::= @T[USEPOINT;WRITE(STATION,F0314);];
<E0315> ::= @T[USEPOINT;WRITE(STATION,F0315);WRITE(STATION,FSEP);];
<E0316> ::= @T[USEPOINT;WRITE(STATION,F0316);];
<E0317> ::= @T[USEPOINT;WRITE(STATION,F0317);];
<E0318> ::= @T[USEPOINT;WRITE(STATION,F0318);WRITE(STATION,FSEP);];
<E0319> ::= @T[USEPOINT;WRITE(STATION,F0319);];
<E0401> ::= @T[USEPOINT;WRITE(STATION,F0401);];
<E0402> ::= @T[USEPOINT;WRITE(STATION,F0402);];
<E0403> ::= @T[USEPOINT;WRITE(STATION,F0403);WRITE(STATION,FSEP);];
<E0404> ::= @T[USEPOINT;WRITE(STATION,F0404);];
<E0405> ::= @T[USEPOINT;WRITE(STATION,F0405);WRITE(STATION,FSEP);];
<E0406> ::= @T[USEPOINT;WRITE(STATION,F0406);];
<E0407> ::= @T[USEPOINT;WRITE(STATION,F0407);];
<E0501> ::= @S[USEPOINT;WRITE(STATION,F0501);];
<E0601> ::= @T[USEPOINT;WRITE(STATION,F0601);];
<E0602> ::= @T[USEPOINT;WRITE(STATION,F0602);WRITE(STATION,FSEP);];
<E0603> ::= @T[USEPOINT;WRITE(STATION,F0603);];
<E0604> ::= @T[USEPOINT;WRITE(STATION,F0604);];
<E0701> ::= @T[USEPOINT;WRITE(STATION,F0701);];
<E0702> ::= @S[USEPOINT;WRITE(STATION,F0702);WRITE(STATION,FSEP);];
<E0703> ::= @T[USEPOINT;WRITE(STATION,F0703);];
<E0801> ::= @T[USEPOINT;WRITE(STATION,F0801);];
<E0802> ::= @T[USEPOINT;WRITE(STATION,F0802);];
<E0803> ::= @T[USEPOINT;WRITE(STATION,F0803);];
<E0804> ::= @S[USEPOINT;WRITE(STATION,F0804);WRITE(STATION,FSEP);];
<E0805> ::= @T[USEPOINT;WRITE(STATION,F0805);];
<E0806> ::= @T[USEPOINT;WRITE(STATION,F0806);];
<E0807> ::= @T[USEPOINT;WRITE(STATION,F0807);WRITE(STATION,FSEP);];
<E0808> ::= @T[USEPOINT;WRITE(STATION,F0808);WRITE(STATION,FSEP);];
<E0809> ::= @T[USEPOINT;WRITE(STATION,F0809);];
<E0901> ::= @S[USEPOINT;WRITE(STATION,F0901);WRITE(STATION,FSEP);];
<E0902> ::= @T[USEPOINT;WRITE(STATION,F0902);];
<E0903> ::= @T[USEPOINT;WRITE(STATION,F0903);];
<E0904> ::= @T[USEPOINT;WRITE(STATION,F0904);];
<E0905> ::= @T[USEPOINT;WRITE(STATION,F0905);];
<E0906> ::= @T[USEPOINT;WRITE(STATION,F0906);];
<E0907> ::= @T[USEPOINT;WRITE(STATION,F0907);];
<E0908> ::= @T[USEPOINT;WRITE(STATION,F0908);];
<E0909> ::= @T[USEPOINT;WRITE(STATION,F0909);];
<E0910> ::= @T[USEPOINT;WRITE(STATION,F0910);];
<E0911> ::= @T[USEPOINT;WRITE(STATION,F0911);];
<E1001> ::= @T[USEPOINT;WRITE(STATION,F1001);];
<E1002> ::= @T[USEPOINT;WRITE(STATION,F1002);];
```

```
<E1003> ::= @T[USEPOINT;WRITE(STATION,F1003);];
<E1004> ::= @T[USEPOINT;WRITE(STATION,F1004);];
<E1005> ::= @T[USEPOINT;WRITE(STATION,F1005);];
<E1006> ::= @T[USEPOINT;WRITE(STATION,F1006);];
<E1007> ::= @S[USEPOINT;WRITE(STATION,F1007);];
<E1101> ::= @T[USEPOINT;WRITE(STATION,F1101);];
<E1102> ::= @T[USEPOINT;WRITE(STATION,F1102);WRITE(STATION,FSEP);];
<E1103> ::= @S[USEPOINT;WRITE(STATION,F1103);];
<E1201> ::= @T[USEPOINT;WRITE(STATION,F1201);];
<E1203> ::= @T[USEPOINT;WRITE(STATION,F1203);WRITE(STATION,FSEP);];
<E1204> ::= @T[USEPOINT;WRITE(STATION,F1204);];
<E1205> ::= @S[USEPOINT;WRITE(STATION,F1205);WRITE(STATION,FSEP);];
<E1206> ::= @T[USEPOINT;WRITE(STATION,F1206);];
<E1207> ::= @T[USEPOINT;WRITE(STATION,F1207);];
<E1211> ::= @T[USEPOINT;WRITE(STATION,F1211);];
<E1212> ::= @T[USEPOINT;WRITE(STATION,F1212);WRITE(STATION,FSEP);];
<E1213> ::= @T[USEPOINT;WRITE(STATION,F1213);];
<E1215> ::= @T[USEPOINT;WRITE(STATION,F1215);WRITE(STATION,FSEP);];
<E1216> ::= @T[USEPOINT;WRITE(STATION,F1216);];
<E1218> ::= @T[USEPOINT;WRITE(STATION,F1218);WRITE(STATION,FSEP);];
<E1219> ::= @T[USEPOINT;WRITE(STATION,F1219);];
<E1220> ::= @T[USEPOINT;WRITE(STATION,F1220);];
<E1222> ::= @T[USEPOINT;WRITE(STATION,F1222);];
<E1223> ::= @T[USEPOINT;WRITE(STATION,F1223);];
<E1224> ::= @T[USEPOINT;WRITE(STATION,F1224);];
<E1226> ::= @T[USEPOINT;WRITE(STATION,F1226);];
<E1227> ::= @T[USEPOINT;WRITE(STATION,F1227);];
<E1228> ::= @T[USEPOINT;WRITE(STATION,F1228);WRITE(STATION,FSEP);];
<E1234> ::= @T[USEPOINT;WRITE(STATION,F1234);];
<E1236> ::= @T[USEPOINT;WRITE(STATION,F1236);];
<E1237> ::= @T[USEPOINT;WRITE(STATION,F1237);];
<E1238> ::= @T[USEPOINT;WRITE(STATION,F1238);WRITE(STATION,FSEP);];
<E1240> ::= @T[USEPOINT;WRITE(STATION,F1240);];
<E1241> ::= @T[USEPOINT;WRITE(STATION,F1241);WRITE(STATION,FSEP);];
<E1242> ::= @T[USEPOINT;WRITE(STATION,F1242);];
<E1243> ::= @T[USEPOINT;WRITE(STATION,F1243);];
<E1244> ::= @T[USEPOINT;WRITE(STATION,F1244);];
<E1245> ::= @T[USEPOINT;WRITE(STATION,F1245);];
<E1246> ::= @T[USEPOINT;WRITE(STATION,F1246);WRITE(STATION,FSEP);];
<E1247> ::= @T[USEPOINT;WRITE(STATION,F1247);];
<E1248> ::= @T[USEPOINT;WRITE(STATION,F1248);WRITE(STATION,FSEP);];
<E1300> ::= @Q[ WRITE (STATION, F1300) ; ] ;
<E1301> ::= @T[USEPOINT;WRITE(STATION,F1301);];
<E1302> ::= @T[USEPOINT;WRITE(STATION,F1302);];
<E1303> ::= @T[USEPOINT;WRITE(STATION,F1303);];
<E1304> ::= @T[USEPOINT;WRITE(STATION,F1304);];
<E1305> ::= @T[USEPOINT;WRITE(STATION,F1305);];
<E1306> ::= @T[USEPOINT;WRITE(STATION,F1306);WRITE(STATION,FSEP);];
<E1307> ::= @T[USEPOINT;WRITE(STATION,F1307);];
<E1308> ::= @T[USEPOINT;WRITE(STATION,F1308);WRITE(STATION,FSEP);];
<E1309> ::= @T[USEPOINT;WRITE(STATION,F1309);];
<E1310> ::= @T[USEPOINT;WRITE(STATION,F1310);];
<E1311> ::= @T[USEPOINT;WRITE(STATION,F1311);];
<PARSE01> ::= @Q[ WRITE (STATION, FNEW) ; ] ;
<PARSE02> ::= @Q[ WRITE (STATION, FPAR) ; ] ;
<PARSE03> ::= @Q[ WRITE (STATION, FSET) ; ] ;
<PARSE04> ::= @Q[ WRITE (STATION, FINT) ; ] ;
```

```
<PARSE05> ::= @Q[ WRITE (STATION, FEND) ; ] ;
<PARSE06> ::= @Q[ WRITE (STATION, FIFS) ; ] ;
<PARSE07> ::= @Q[ WRITE (STATION, FIUS) ; ] ;
<PARSE08> ::= @Q[ WRITE (STATION, FFOR) ; ] ;
<PARSE09> ::= @Q[ IF ALTERNATIVE = 1 THEN WRITE (STATION, FITE) ; ] ;
<PARSE10> ::= @Q[ WRITE (STATION, FDEB) ; ] ;
<PARSE11> ::= @Q[ WRITE (STATION, FPRO) ; ] ;
<PARSE12> ::= @Q[ WRITE (STATION, FASS) ; ] ;
DONT BACKSUBSTITUTE <NEWOL2BLOCK>, <PARTITIONSTATEMENT>, <SETSTATEMENT>,
     <INTERCHANGESTATEMENT>, <ENDSTATEMENT>, <DEBUGSTATEMENT>, <OL2IFS>,
     <OL2PROCEDURESTATEMENT>, <OL2ITERATIVECLAUSE>, <OL2FORSTATEMENT>,
     <OL2IOSTATEMENT>, <ASSIGNMENTSTATEMENT>, <FINISH>, <PL1STATEMENT> ;
RESERVE  FINISH, LET, DEFINE, DENOTE, PARTITION, SET, INTERCHANGE,
         IF, INPUT, OUTPUT, FOR, WHILE, UNTIL, DEBUG, NODE, PRINT,
         PROCEDURE, PROC , AND, AFTER, NULL ;
  END.




COMMENT  4"00" INSERTS A HEX ZERO INTO THE OUTPUT LINE.  ITS PURPOSE IS
        TO PREVENT THE WRITING OF MORE THAN 74 CHARACTERS TO A LINE AT
        THE TERMINAL, WHICH CAUSES DOUBLE SPACING.  IT ALSO PRINTS AS A
        '?' ON THE LINE PRINTER.
         4"4D" IS THE CHARACTER (
         4"5D" IS THE CHARACTER )
         4"7D" IS THE CHARACTER '
        IT IS NECESSARY TO USE HEX CODING BECAUSE OF THE LIMITATIONS
        OF THE FORMAT STATEMENT IN ALGOL ON THE B6500 ;
FORMAT FSEP ("      POSSIBLE INCORRECT SEPARATOR ELEMENT IN PRECEDING SYN
TACTIC UNIT",4"00"),
  F0101 ("<NEWOL2BLOCK>: TERMINAL ';' MISSING",4"00"),
  F0102 ("<IDENTIFIERDECLARATION>: INCORRECT <IDLIST>",4"00"),
  F0103 ("<IDENTIFIERDECLARATION>: INCORRECT <DEFINITION>",4"00"),
  F0104 ("<IDLIST>: INCORRECT <IDENTIFIER>",4"00"),
  F0105 ("<DEFINITION>: 'DIMENSIONAL' MISSING FOLLOWING 'FINITE'",
4"00"),
  F0106 ("<DEFINITION>: 'SPACE<S>' MISSING FOLLOWING 'VECTOR'",4"00"),
  F0107 ("<DEFINITION>: 'OF' MISSING FOLLOWING 'SPACE<S>'",4"00"),
  F0108 ("<DEFINITION>: 'DIMENSION' MISSING FOLLOWING 'OF'",4"00"),
  F0109 ("<DEFINITION>: INCORRECT <NBYN> FOLLOWING 'DIMENSION'",4"00"),
  F0110 ("<DEFINITION>: 'HAS' OR 'CONTAINS' MISSING FOLLOWING 'WHICH'",
4"00"),
  F0111 ("<DEFINITION>: 'AS' MISSING FOLLOWING 'HAS' OR 'CONTAINS'",
4"00"),
  F0112 ("<DEFINITION>: 'ELEMENT<S>' OR 'MEMBER<S>' MISSING FOLLOWING 'T
HE'",4"00"),
  F0113 ("<NBYN>: INCORRECT <BOUNDPAIREXPRESSION> FOLLOWING ",
4"7D4D7D00"),
  F0114 ("<NBYN>: ",4"7D5D7D"," MISSING FOLLOWING <BOUNDPAIREXPRESSION>"
,4"00"),
  F0115 ("<SEQUENCE>: 'ARE' OR 'IS A' MISSING FOLLOWING 'WHICH'",4"00"),
  F0116 ("<MODULUS>: ",4"7D5D7D"," MISSING FOLLOWING LIST OF <*N>",
4"00"),
  F0117 ("<MODULUS>: INCORRECT <*N> IN LIST",4"00"),
  F0119 ("<MODULUS>: '",4"5D",''' MISSING FOLLOWING LIST OF <*N>",4"00"),
  F0120 ("<OTHERATTRIBUTES>: INCORRECT <ATTRIBUTE> IN LIST",4"00"),
  F0121 ("<OTHERATTRIBUTES>: RIGHT PARENTHESIS MISSING FOLLOWING LIST OF
 <ATTRIBUTE>",4"00"),
  F0122 ("<PRECISION>: INCORRECT <*N> FOLLOWING ",4"7D4D7D00"),
  F0123 ("<PRECISION>: INCORRECT <*N> FOLLOWING ','",4"00"),
  F0124 ("<PRECISION>: RIGHT PARENTHESIS MISSING FOLLOWING LIST OF <*N>"
,4"00"),
  F0125 ("<DIMENSION>: INCORRECT <NBYN> FOLLOWING 'BOUND<S>' OR 'ORDER'"
,4"00"),
  F0128 ("<BLOCKOF>: 'BLOCK<S>' MISSING FOLLOWING FIRST <SEQUENCE>",
4"00"),
  F0129 ("<BLOCKOF>: INCORRECT <SEQUENCE> FOLLOWING 'BLOCK<S>'",4"00"),
  F0130 ("<BLOCKOF>: 'OF' MISSING FOLLOWING SECOND <SEQUENCE>",4"00"),
  F0131 ("<BLOCKOF>: INCORRECT <*I> FOLLOWING 'OF'",4"00"),
  F0132 ("<ONORINASPACE>: INCORRECT <*I> FOLLOWING 'ON'",4"00"),
  F0133 ("<ONORINASPACE>: INCORRECT <SPACEID> IN LIST FOLLOWING 'FROM'",
4"00"),
  F0135 ("<ONORINASPACE>: 'INTO' OR 'ONTO' OR 'TO' MISSING FOLLOWING LIS
T",4"00"/"FOLLOWING 'FROM'",4"00"),
  F0136 ("<ONORINASPACE>: INCORRECT <SPACEID> IN LIST FOLLOWING 'INTO' O
R",4"00"/"'ONTO' OR 'TO'",4"00"),
  F0138 ("<ONORINASPACE>: 'OF' OR 'IN' MISSING FOLLOWING 'ELEMENT<S>' OR
 'MEMBER<S>'",4"00"),
  F0139 ("<ONORINASPACE>: INCORRECT <*I> FOLLOWING 'OF' OR 'IN'",4"00"),
  F0140 ("<SPACEID>: INCORRECT <*I> FOLLOWING ",4"7D4D7D00"),
  F0141 ("<SPACEID>: '",4"5D","' MISSING FOLLOWING <*I>",4"00"),
  F0142 ("<SPACEID>: INCORRECT <*I>",4"00"),
  F0201 ("<PARTITIONSTATEMENT>: INCORRECT <ROOTLIST> FOLLOWING 'PARTITIO
N'",4"00"),
  F0202 ("<PARTITIONSTATEMENT>: 'AFTER' MISSING FOLLOWING <ROOTLIST>",
4"00"),
  F0203 ("<PARTITIONSTATEMENT>: INCORRECT <ROWORCOLUMN> FOLLOWING 'AFTER
'",4"00"),
  F0204 ("<PARTITIONSTATEMENT>: INCORRECT <PL1EXPRESSION> IN FIRST LIST"
,4"00"),
  F0206 ("<PARTITIONSTATEMENT>: INCORRECT <ROWORCOLUMN> FOLLOWING 'AND'"
,4"00"),
  F0207 ("<PARTITIONSTATEMENT>: INCORRECT <PL1EXPRESSION> IN SECOND LIST
",4"00"),
  F0209 ("<PARTITIONSTATEMENT>: TERMINAL ';' MISSING",4"00"),
  F0210 ("<ROOTLIST>: INCORRECT <ROOTPARTSEQ> IN LIST",4"00"),
  F0301 ("<SETSTATEMENT>: INCORRECT <SIMPLESETSTMT> IN LIST FOLLOWING 'S
ET'",4"00"),
  F0303 ("<SETSTATEMENT>: TERMINAL ';' MISSING",4"00"),
  F0304 ("<SIMPLESETSTMT>: INCORRECT <IDENTIFIER>",4"00"),
  F0305 ("<SIMPLESETSTMT>: INCORRECT <SECTION> FOLLOWING 'THE'",4"00"),
  F0306 ("<SIMPLESETSTMT>: 'PART' MISSING FOLLOWING <SECTION>",4"00"),
  F0307 ("<SIMPLESETSTMT>: 'OF' MISSING FOLLOWING 'PART'",4"00"),
  F0308 ("<SIMPLESETSTMT>: INCORRECT <ROOTPARTSEQ> FOLLOWING 'OF'",
4"00"),
  F0309 ("<SIMPLESETSTMT>: INCORRECT <TYPEDIDENT> FOLLOWING 'EQUAL' OR '
=' OR 'TO'",4"00"),
  F0310 ("<TYPEDIDENT>: INCORRECT <ROOTPARTSEQ>",4"00"),
  F0312 ("<ROOTPARTSEQ>: INCORRECT <PL1EXPRESSION> IN LIST FOLLOWING '['
",4"00"),
  F0313 ("<ROOTPARTSEQ>: ']' MISSING FOLLOWING LIST",4"00"),
  F0314 ("<ROOTPARTSEQ>: INCORRECT <PL1EXPRESSION> IN LIST FOLLOWING '<'
",4"00"),
  F0315 ("<ROOTPARTSEQ>: '>' MISSING FOLLOWING LIST",4"00"),
  F0316 ("<IDENTIFIER>: INCURRECT <*I>",4"00"),
  F0317 ("<IDENTIFIER>: INCURRECT <PL1EXPRESSION> FOLLUWING '['",4"00"),
  F0318 ("<IDENTIFIER>: ']' MISSING FOLLOWING <PL1EXPRESSIUN>",4"00"),
  F0319 ("<SECTION>: 'ADJOINT' UR 'ADJ' MISSING FOLLOWING 'SELF'",
4"00"),
  F0401 ("<INTERCHANGESTATEMENT>: INCORRECT <ROWORCULUMN> FOLLOWING 'INT
ERCHANGE'",4"00"),
  F0402 ("<INTERCHANGESTATEMENT>: INCORRECT <PL1EXPRESSION> FOLLOWING <R
OWORCULUMN>",4"00"),
  F0403 ("<INTERCHANGESTATEMENT>: 'AND' MISSING FOLLOWING FIRST <PL1EXPR
ESSION>",4"00"),
  F0404 ("<INTERCHANGESTATEMENT>: INCORRECT <PL1EXPRESSION> FOLLOWING 'A
ND'",4"00"),
  F0405 ("<INTERCHANGESTATEMENT>: 'IN' OR 'OF' OR 'ON' MISSING FOLLOWING
 SECOND",4"00"/"<PL1EXPRESSION>",4"00"),
  F0406 ("<INTERCHANGESTATEMENT>: INCURRECT <ROOTPARTSEQ>",4"00"),
  F0407 ("<INTERCHANGESTATEMENT>: TERMINAL ';' MISSING UR INCORRECT <ROO
TPARTSEQ>",4"00"),
  F0501 ("<ENDSTATEMENT>: TERMINAL ';' MISSING OR INCORRECT <*I>",
4"00"),
  F0601 ("<OL2IFS>: INCORRECT <OL2BOOLEANEXP> FOLLOWING 'IF'",4"00"),
  F0602 ("<OL2IFS>: 'THEN' MISSING FOLLOWING <OL2BOOLEANEXP>",4"00"),
  F0603 ("<OL2IFS>: INCURRECT <STATEMENT> FOLLOWING 'THEN'",4"00"),
  F0604 ("<OL2IFS>: INCURRECT <STATEMENT> FULLOWING 'ELSE'",4"00"),
  F0701 ("<OL2IOSTATEMENT>:  INCORRECT <LISTOFOPS> FOLLUWING 'INPUT' OR
'OUTPUT'",4"00"),
  F0702 ("<OL2IOSTATEMENT>: TERMINAL ';' MISSING",4"00"),
  F0703 ("<LISTUFOPS>: INCURRECT <*I> IN LIST",4"00"),
  F0801 ("<OL2FURSTATEMENT>: INCORRECT <STEPPEDVARIABLE> FULLOWING 'FOR'
",4"00"),
  F0802 ("<OL2FURSTATEMENT>: '=' MISSING FOLLOWING <STEPPEDVARIABLE>",
4"00"),
  F0803 ("<OL2FORSTATEMENT>:  INCORRECT <SPECIFICATION> FOLLOWING '='",
4"00"),
  F0804 ("<OL2FURSTATEMENT>: TERMINAL ';' MISSING",4"00"),
  F0805 ("<SPECIFICATIUN>: INCORRECT INITIAL <UNIT>",4"00"),
  F0806 ("<SPECIFICATIUN>: ',' MISSING FOLLOWING FIRST <UNIT>",4"00"),
  F0807 ("<SPECIFICATIUN>: INCURRECT <UNIT> FOLLOWING ','",4"00"),
  F0808 ("<SPECIFICATIUN>: '..' MISSING FOLLOWING LAST <UNIT>",4"00"),
  F0809 ("<SPECIFICATIUN>: INCORRECT TERMINAL <UNIT> FULLOWING '..'",
4"00"),
  F0901 ("<OL2ITERATIVECLAUSE>: TERMINAL ';' MISSING",4"00"),
  F0902 ("<CLAUSE>: INCURRECT <UL2BUOLEANEXP> FOLLOWING 'WHILE' OR 'UNTI
L'",4"00"),
  F0903 ("<OL2BUOLEANEXP>: INCORRECT <BOOLEANTERM> IN LIST",4"00"),
  F0904 ("<BOOLEANTERM>: INCURRECT <BOOLEANFACTOR> IN LIST",4"00"),
  F0905 ("<BOOLEANFACTUR>: INCORRECT <OL2BOOLEANEXP> FULLOWING ",
4"7D40/D00"),
  F0906 ("<BOOLEANFACTUR>: '",4"50",'' MISSING FOLLOWING <UL2BOOLEANEXP>
",4"00"),
  F0907 ("<BOOLEANFACTUR>: INCURRECT <BOOLEANFACTOR> FOLLOWING '¬'",
4"00"),
  F0908 ("<BOOLEANFACTUR>: INCORRECT <COMPAREEXPRESSION>",4"00"),
  F0909 ("<BOOLEANFACTUR>: INCURRECT <COMPAREEXPRESSION> FULLOWING <COMP
AREOP>",4"00"),
  F0910 ("<OL2BOOLEANEXP>: INCORRECT <BOOLEANTERM> FOLLOWING '|'",
4"00"),
  F0911 ("<BOOLEANTERM>: INCORRECT <BOOLEANFACTOR> FOLLOWING '&'",
4"00"),
  F1001 ("<DEBUGSTATEMENT>: 'TABLE' MISSING FOLLOWING 'ID'",4"00"),
  F1002 ("<DEBUGSTATEMENT>: 'ID' OR 'TREE' MISSING FOLLOWING 'PRINT'",
4"00"),
  F1003 ("<DEBUGSTATEMENT>: 'NODES' MISSING FOLLOWING 'TREE'",4"00"),
  F1004 ("<DEBUGSTATEMENT>: 'PRINT' MISSING FOLLOWING 'NODE'",4"00"),
  F1005 ("<DEBUGSTATEMENT>: 'OFF' MISSING FOLLOWING 'PRINT'",4"00"),
  F1006 ("<DEBUGSTATEMENT>: 'ON' OR 'OFF' MISSING FOLLOWING 'TRACE'",
4"00"),
  F1007 ("<DEBUGSTATEMENT>: TERMINAL ';' MISSING",4"00"),
  F1101 ("<OL2PROCEDURESTATEMENT>: INCORRECT <ARGUMENT> IN LIST",4"00"),
  F1102 ("<OL2PROCEDURESTATEMENT>: '",4"5D","' MISSING",4"00"),
  F1103 ("<OL2PROCEDURESTATEMENT>: TERMINAL ';' MISSING",4"00"),
  F1201 ("<ASSIGNMENTSTATEMENT>: INCORRECT <OL2LEFTHANDSIDE>",4"00"),
  F1203 ("<ASSIGNMENTSTATEMENT>: '=' OR '<-' MISSING",4"00"),
  F1204 ("<ASSIGNMENTSTATEMENT>: INCORRECT <COMPAREEXPRESSION>",4"00"),
  F1205 ("<ASSIGNMENTSTATEMENT>: TERMINAL ';' MISSING",4"00"),
  F1206 ("<REFERENCE>: INCORRECT <BASICREF> FOLLOWING '->'",4"00"),
  F1207 ("<BASICREF>: INCORRECT <UNQUAL> FOLLOWING '.'",4"00"),
  F1211 ("<UNQUAL>: INCORRECT <PL1EXPRESSION> FOLLOWING '['",4"00"),
  F1212 ("<UNQUAL>: ']' MISSING FOLLOWING <PL1EXPRESSION>",4"00"),
  F1213 ("<UNQUAL>: INCORRECT <PL1EXPRESSION> IN LIST FOLLOWING '<'",
4"00"),
  F1215 ("<UNQUAL>: '>' MISSING FOLLOWING LIST OF <PL1EXPRESSION>",
4"00"),
  F1216 ("<UNQUAL>: INCORRECT <OL2ARITHMETICEXPRESSION> IN LIST",4"00"),
  F1218 ("<UNQUAL>: '",4"5D","' MISSING FOLLOWING LIST OF <OL2ARITHMETIC
EXPRESSION>",4"00"),
  F1219 ("<OL2ARITHMETICEXPRESSION>: INCORRECT <OL2TERM> FOLLOWING '+' O
R '-'",4"00"),
  F1220 ("<OL2TERM>: INCORRECT <OL2DIVIDE> FOLLOWING '*'",4"00"),
  F1222 ("<OL2DIVIDE>: INCORRECT <OL2PRIMARY> FOLLOWING '/'",4"00"),
  F1223 ("<OL2FACTOR>: INCORRECT <OL2PRIMARY>",4"00"),
  F1224 ("<OL2FACTOR>: INCORRECT <OL2EXTRA> FOLLOWING '**'",4"00"),
  F1226 ("<OL2EXTRA>: INCORRECT <OL2EXTRA> FOLLOWING '**'",4"00"),
  F1227 ("<MODIFIEDEXPRESSIONUNIT>: INCORRECT <OL2ARITHMETICEXPRESSION>"
,4"00"),
  F1228 ("<MODIFIEDEXPRESSIONUNIT>: '",4"5D","' MISSING",4"00"),
  F1229 ("<MODIFIEDOL2IDENTIFIER>: INCORRECT <PL1EXPRESSION> FOLLOWING '
['",4"00"),
  F1230 ("<MODIFIEDOL2IDENTIFIER>: ']' MISSING FOLLOWING <PL1EXPRESSION>
",4"00"),
  F1231 ("<MODIFIEDOL2IDENTIFIER>: INCORRECT <PL1EXPRESSION> IN LIST FOL
LOWING '<'",4"00"),
  F1232 ("<MODIFIEDOL2IDENTIFIER>: '>' MISSING",4"00"),
  F1234 ("<BASICS>: '",4"5D","' MISSING FOLLOWING <BASICS>",4"00"),
  F1236 ("<INNERPRODUCT>: ',' MISSING FOLLOWING <OL2ARITHMETICEXPRESSION
>",4"00"),
  F1237 ("<INNERPRODUCT>: INCORRECT <OL2ARITHMETICEXPRESSION> FOLLOWING
','",4"00"),
  F1238 ("<INNERPRODUCT>: RIGHT PARENTHESIS MISSING FOLLOWING SECOND",
4"00",/"<OL2ARITHMETICEXPRESSION>",4"00"),
  F1240 ("<NORM>: INCORRECT <OL2ARITHMETICEXPRESSION> FOLLOWING '|'",
4"00"),
  F1241 ("<NORM>: '||' MISSING FOLLOWING <OL2ARITHMETICEXPRESSION>",
4"00"),
  F1242 ("<CONSTANT>: '+' OR '-' MISSING FOLLOWING 'E'",4"00"),
  F1243 ("<CONSTANT>: INCORRECT <*N> IN EXPONENT",4"00"),
  F1244 ("<BOUNDPAIREXPRESSION>: '",4"4D",",' MISSING OR INCORRECT <*N>",
4"00"),
  F1245 ("<BOUNDPAIREXPRESSION>: INCORRECT <PL1EXPRESSION> IN LIST FOLLO
WING ",4"7D407D00"),
  F1246 ("<BOUNDPAIREXPRESSION>: ':' MISSING FOLLOWING <PL1EXPRESSION> I
N LIST",4"00"),
  F1247 ("<BOUNDPAIREXPRESSION>: INCORRECT <PL1EXPRESSION> IN LIST FOLLO
WING ':'",4"00"),
  F1248 ("<BOUNDPAIREXPRESSION>: '",4"5D",",' MISSING",4"00"),
  F1300 ("<PL1STATEMENT> ENCOUNTERED, NO SYNTAX PARSING PERFORMED",
4"00"),
  F1301 ("<PL1EXPRESSION>: INCORRECT <PL1TERM> FOLLOWING '+' OR '-'",
4"00"),
  F1302 ("<PL1TERM>: INCORRECT <PL1FACTOR> FOLLOWING '*' OR '/'",4"00"),
  F1303 ("<PL1FACTOR>: INCORRECT <PL1FACTOR> FOLLOWING UNARY '+' OR '-'"
,4"00"),
  F1304 ("<PL1FACTOR>: INCORRECT <PL1FACTOR> FOLLOWING '**'",4"00"),
  F1305 ("<PL1BASIC>: INCORRECT <PL1EXPRESSION> FOLLOWING ",
4"7D407D00"),
  F1306 ("<PL1BASIC>: '",4"5D",",' MISSING FOLLOWING <PL1EXPRESSION>",
4"00"),
  F1307 ("<PL1UNQUAL>: INCORRECT <PL1EXPRESSION> OR '*' IN LIST",4"00"),
  F1308 ("<PL1UNQUAL>: '",4"5D",",' MISSING FOLLOWING LIST",4"00"),
  F1309 ("<PL1CONSTANT>: INCORRECT <*N> IN EXPONENT",4"00"),
  F1310 ("<PL1REFERENCE>: INCORRECT <PL1BASICREF> FOLLOWING '->'",
4"00"),
  F1311 ("<PL1BASICREF>: INCORRECT <PL1UNQUAL> FOLLOWING '.'",4"00"),
  FNEW ("STATEMENT PARSES AS <NEWOL2BLOCK>--ANY ERRORS NOTED BELOW",
4"00"),
  FPAR ("STATEMENT PARSES AS <PARTITIONSTATEMENT>--ANY ERRORS NOTED BELO
W",4"00"),
  FSET ("STATEMENT PARSES AS <SETSTATEMENT>--ANY ERRORS NOTED BELOW",
4"00"),
  FINT ("STATEMENT PARSES AS <INTERCHANGESTATEMENT>--ANY ERRORS NOTED BE
LOW",4"00"),
  FEND ("STATEMENT PARSES AS <ENDSTATEMENT>--ANY ERRORS NOTED BELOW",
4"00"),
  FIFS ("STATEMENT PARSES AS <OL2IFS>--ANY ERRORS NOTED BELOW",4"00"),
  FIOS ("STATEMENT PARSES AS <OL2IOSTATEMENT>--ANY ERRORS NOTED BELOW",
4"00"),
  FFOR ("STATEMENT PARSES AS <OL2FORSTATEMENT>--ANY ERRORS NOTED BELOW",
4"00"),
  FITE ("STATEMENT PARSES AS <OL2ITERATIVECLAUSE>--ANY ERRORS NOTED BELO
W",4"00"),
  FDEB ("STATEMENT PARSES AS <DEBUGSTATEMENT>--ANY ERRORS NOTED BELOW",
4"00"),
  FPRO ("STATEMENT PARSES AS <OL2PROCEDURESTATEMENT>--ANY ERRORS NOTED B
ELOW",4"00"),
  FASS ("STATEMENT PARSES AS <ASSIGNMENTSTATEMENT>--ANY ERRORS NOTED BEL
OW",4"00");
```

APPENDIX C

DOCUMENTATION OF OL2/SKELETON

The TWST65 compiler-compiler available on the Burroughs 6500 computer system currently in Coordinated Science Laboratory uses two input files, known internally to TWST65 as CARD and SKELETON. The CARD file is expected to contain the syntax and semantic routines for the user's project, as described by Trout (1, 2). OL2/TWST, the file used for that purpose in generating OL2/PARSER, is described in Appendix B. The SKELETON file is expected to provide the basic procedures for the parsing mechanism, such as input and output, string scanning, stack manipulation, and table building and searching.

The TWST65 system contains a standard file, TWST65/SKELETON, which contains these procedures; however, a program generated with these routines included will expect card input with merged patches from tape files, and it will parse the entire input file as a single syntactic unit (usually a program).

OL2/SKELETON is the modification of TWST65/SKELETON which was used in generating OL2/PARSER, which uses remote terminal files for input and output, and which scans each new input string reentrantly. That is, OL2/PARSER parses input characters from the attached terminal until either it recognizes a complete OL/2 statement or it finds a syntactic error, at which point it starts over.

Interactive Features

The first modifications, those to make OL2/PARSER interactive, are contained in the procedure READALINE.

READALINE is substituted for the procedure READACARD found in TWST65/SKELETON. It is the only procedure in OL2/PARSER which accepts input, and it accepts data from the file STATION. Whenever a remote terminal user on the B6500 executes a program containing a file named STATION, the data communication system will equate his terminal with that file. Since all output from OL2/PARSER except a final listing of statements and error markers is written to the file STATION, OL2/PARSER is interactive.

READALINE (at line 48200 in the listing) fills the one dimensional alpha array CARDBUF with the characters entered at the terminal, places a %-character in the last position of the array (to prevent the scanning routines from scanning beyond the array and to signal when more characters are needed), and initializes NCR (an integer variable which is used to point to positions in CARDBUF). READALINE also causes error markers for incorrect statements to be placed under the incorrect statements in the printer output file, LINE, and inserts the new input string into LINE. The Boolean variable USEPT is set TRUE to cause the procedure USEPOINT (discussed later) to adjust the error marker when errors are found.

READALINE can be invoked from only two procedures in OL2/PARSER, ENDOFPROGRAMDEFINE and NEXTCHAR (at lines 88900 and

51400 of OL2/SKELETON). ENDOFPROGRAMDEFINE calls READALINE whenever a new statement is needed, and NEXTCHAR calls READALINE whenever more input is needed to determine whether a current string is a complete, valid statement.

Reentrant Scanning

The second modifications, those to cause OL2/PARSER to parse each new string independently, are contained in ENDOFPROGRAMDEFINE and TWSTINITIAL. It is much easier to let STATEMENT be the largest syntactic unit and restart the parse for each statement than to let PROGRAM be the largest syntactic unit and set program parameters back to previous values when errors are encountered.

A TWST65 generated program has only one significant executable statement, that which calls an initial procedure. All work is accomplished via calls to procedures which scan strings, manipulate stacks, use the identifier table, and recognize various syntactic units. The procedures are reentrant to allow recursive parsing.

The simplest way to handle errors is to identify the error as precisely as possible, terminate the parse of that string, call for more input, and start over with the new data. ENDOFPROGRAMDEFINE and TWSTINITIAL were rewritten to achieve this method of operation. The only key change is explicitly setting an integer variable LRS to zero for each input string. LRS is a stack variable used in manipulating recognized syntactic units. Setting it to zero is the same as emptying the stack. LRS is the only variable that was not explicitly initialized before use in TWST65/SKELETON.

ENDOFPROGRAMDEFINE (line 88900) contains initialization for the program, including setting up the warning message for undeclared identifiers and the heading for the listing of input strings and error markers produced at the end of the run. MASTERBOOLEAN is a word of storage whose bits are each used as Boolean variables. FIRSTCOL and LASTCOL define the maximum length of a string accepted as data; 74 is used because the Hazeltine CRT's in the B6500 room have screens 74 characters wide. NUMERRS is an integer variable used to count errors (maximum of one per statement). Setting FINISH to FALSE allows the program to begin normally.

The central loop of OL2/PARSER is contained in this procedure (lines 90400-90900). Until the word FINISH is entered by the user, setting the Boolean variable FINISH to TRUE, OL2/PARSER will execute this loop. Each time through the loop, the program will prepare to parse a new statement, read a line from the terminal, parse that input, and display a message describing whether the string was completely parsed.

The initialization for a new statement takes place in TWSTINITIAL (lines 33100-33800). Here LRS is zeroed, and several variables (LSPACES, BASE, ENTERTOGGLE, CBPOI, and SYLPH) are set as they are in TWST65/SKELETON. ALTERNATIVE is an integer variable which is used within the syntax of OL2/PARSER to control some alternatives in the grammer (see

Appendix B for a discussion of the syntax of OL2/PARSER).

Input from the terminal is discussed earlier in the description of READALINE.

The parse is initiated by the variable CALL, which is defined by TWST65 to equal TESTSTATEMENT(1). TESTSTATEMENT is the TWST65 generated Boolean procedure which scans the input string for an occurrence of the syntactic unit STATEMENT as defined in OL2/TWST (see Appendix B).

Since CALL = TESTSTATEMENT(1), the scanner begins at the start of the input buffer with an empty stack. TESTSTATEMENT returns the value of TRUE if a sequence of syntactic alternatives is encountered which allows the parse for a STATEMENT to be satisfied. Displaying an error message is often an alternative within the syntax of OL2/TWST, so the message "PARSE COMPLETE" does not mean that no errors exist in an input string.

TESTSTATEMENT returns the value FALSE when the parser cannot find an allowed alternative before reaching the end of the input string. Most error messages will cause the parser to backtrack in order to force a rapid termination of the parse. Otherwise, numerous useless error messages might be listed. (Appendix B contains discussion of this feature.)

The final modification of TWST65/SKELETON in the construction of OL2/SKELETON is the procedure USEPOINT (line 86700). This procedure is called whenever an error is found in an input string. If the Boolean variable USEPT is TRUE, then the first error in the input string has been encountered; otherwise, the error marker has already been positioned for this input string. If USEPT is TRUE, it is set FALSE and the alpha array POINT is filled with hexadecimal zeroes, which cause termination of a line of remote terminal output when encountered. Then, blanks are inserted in POINT up to the position of the first character not scanned, where the symbol "<" is inserted as an error marker to be displayed on the terminal and in the listing of statements. Finally, the error count is increased. SEMANTICTEST is always given the value FALSE to force the parse to fail at this alternative.

File Listing

The file OL2/SKELETON is maintained as shown in Appendix D; a listing follows.

```
COMMENT
  SCANNER AND UTILITY ROUTINES FOR THE INTERACTIVE OL/2 SYNTAX
  ANALYZER WRITTEN FOR PROFESSOR PHILLIPS OF DCS.

  THIS FILE IS A MODIFICATION OF TWST65/SKELETON, WHICH PROVIDES THE
  BASIC ROUTINES FOR COMPILERS PRODUCED BY THE TWST65 COMPILER-
  COMPILER ON THE B6500 SYSTEM.  THE BASIC SCANNING, STACK HANDLING,
  AND TABLE MANIPULATING ROUTINES ARE TOTALLY UNCHANGED.

  THE OPTIONAL PROCEDURES USUALLY PROVIDED BY TWST, LISTPROCEDURES,
  COPY, HEADING, DUMPER, AND PASS2, ARE NOT INCLUDED IN OL2/SKELETON.

  IN ADDITION, READACARD HAS BEEN REPLACED BY READALINE,
  ENDOFPROGRAMDEFINE AND TWSTINITIAL HAVE BEEN REWRITTEN,
  AND USEPOINT HAS BEEN INSERTED.



 BEGIN
VALUE ARRAY LOOKAGAIN(
4"000000000000",
4"000000000000",
4"0000802F803F",
4"0000C01F0036",
4"000000000000",
4"000000000000",
4"000000000000",
4"000000000001");
VALUE ARRAY ALFA(
4"000000000000",
4"000000000000",
4"000000000000",
4"000000000000",
4"000000000000",
4"000000000000",
4"00007FC07FC0",
4"00003FC00000");
VALUE ARRAY ALFANUM(
4"000000000000",
4"000000000000",
4"000000000000",
4"000000000000",
4"000000000000",
4"000000000000",
4"00007FC07FC0",
4"00003FC0FFC0");
VALUE ARRAY WEIGHTS(
4"00003E3F3F3E",
4"00003E3F3F3F",
4"00003E3F3F3F",
4"00003E3F3F3E",
4"00003E3F3E3E",
4"00003E3F3E3E",
4"00003E3F3F3E",
4"00003E3E3E3E",
4"00003E3F3F3E",
4"00003E3F3F3E",
4"00003F3F3F3F",                                                     00005800
4"00003F3F3F3F",                                                     00005900
4"00003F3F3F3F",                                                     00006000
4"00003F3F3F3F",                                                     00006100
4"00003F3F3F3F",                                                     00006200
4"00003F3F3F3F",                                                     00006300
4"00003F3F3F3F",                                                     00006400
4"00003F3F3F3F",                                                     00006500
4"00003F3F3F3F",                                                     00006600
4"00003F3F3F3F",                                                     00006700
4"00003F3F3F3F",                                                     00006800
4"00003F3F3F3F",                                                     00006900
4"00003F3F3F3F",                                                     00007000
4"00003F3F3F3F",                                                     00007100
4"00003F3F3F3F",                                                     00007200
4"00003F3F3F3F",                                                     00007300
4"00003F3F3F3F",                                                     00007400
4"00003F3F3F3F",                                                     00007500
4"00003F3F3F3F",                                                     00007600
4"00003F3F3F3F",                                                     00007700
4"00003F3F3F3F",                                                     00007800
4"00003F3F3F3F",                                                     00007900
4"00003F3F3F3F",                                                     00008000
4"00003F3F3F3F",                                                     00008100
4"00003F3F3F3F",                                                     00008200
4"00003F3F3F3F",                                                     00008300
4"00003F3F3F3F",                                                     00008400
4"00003F3F3F3F",                                                     00008500
4"00003F3F3F3F",                                                     00008600
4"00003F3F3F3F",                                                     00008700
4"00003F3F3F3F",                                                     00008800
4"00003F3F3F3F",                                                     00008900
4"00003F3F3F3F",                                                     00009000
4"00003F3F3F3F",                                                     00009100
4"00003F3F3F3F",                                                     00009200
4"00003F3F3F3F",                                                     00009300
4"00003F3F3F3F",                                                     00009400
4"00003F3F3F3F",                                                     00009500
4"00003F0A0B0C",                                                     00009600
4"00000D0E0F10".                                                     00009700
4"000011123F3F",                                                     00009800
4"00003F3F3F3F",                                                     00009900
4"00003F131415",                                                     00010000
4"000016171819",                                                     00010100
4"00001A1B3F3F",                                                     00010200
4"00003F3F3F3F",                                                     00010300
4"00003F3F1D1E",                                                     00010400
4"00001F202122",                                                     00010500
4"000023243F3F",                                                     00010600
4"00003F3F3F3F",                                                     00010700
4"000000010203",                                                     00010800
4"000004050607",                                                     00010900
4"000008093F3F",                                                     00011000
4"00003F3F3F3F");                                                    00011100
VALUE ARRAY HEXT("0123","4567","89AB","CDEF",0,0,0,0);              00011200
 DEFINE SCANNERVERSION = "23.1,01"#;                                 00011300
 INTEGER                                                             00013500
```

```
          NUMERRS,                                                         00013600
          CARDCOUNT,                                                       00013700
          CARDNUMBER,                                                      00013800
          FIRSTCOL, % =1 FIRST LINE COLUMN TO BE SCANNED                   00013900
          LASTCOL,  % =74 LAST LINE COLUMN TO BE SCANNED                   00014000
          NCR,                                                             00014100
          RSWDBND,   % IF THE RESERVED WORD OPTION IS TAKEN,               00014200
                     % THIS MUST BE SET TO THE LOCATION OF THE             00014300
                     % LAST RESERVED WORD IN BIGTAB (IDTAB)                00014400
          SCANMODE,                                                        00014500
          BTABPTR,                                                         00014600
          BASE,                                                            00014700
          WRK,                                                             00014800
          FIRST,                                                           00014900
          LAST,                                                            00015000
          LRS,                                                             00015100
          ALTERNATIVE, Q, R,                                               00015200
          LSPACES,                                                         00015400
          CHARCOUNT;                                                       00015500
 REAL                                                                      00015600
         TWSTI,                                                            00015700
         I,J,K,                                                            00015800
         MANTISSA,                                                         00015900
         EXPONENT ;                                                        00016000
    ALPHA                                                                  00016100
         NAMEQ;                                                            00016200
                                                                           00016300
DEFINE NXTCHR = (CRPOI + (NCR - 1)) # ;                                    00016400
                                                                           00016500
 ARRAY SLOWUSE[0:31] ;                                                     00016600
 DEFINE                                                                    00016900
          TTIME         =      SLOWUSE[ 00 ]#,                             00017000
          START         =      SLOWUSE[ 01 ]#,                             00017100
          PRTIME        =      SLOWUSE[ 02 ]#,                             00017200
          IOTIME        =      SLOWUSE[ 03 ]#;                             00017300
  BOOLEAN MASTERBOOLEAN, SEMANTICTEST, FINISH, USEPT ;                     00017400
 DEFINE                                                                    00017500
          TRACE0        =      MASTERBOOLEAN.[ 0:1]#,                      00017600
          TRACE1        =      MASTERBOOLEAN.[ 1:1]#,                      00017700
          TRACE2        =      MASTERBOOLEAN.[ 2:1]#,                      00017800
          TRACE3        =      MASTERBOOLEAN.[ 3:1]#,                      00017900
          TRACE4        =      MASTERBOOLEAN.[ 4:1]#,                      00018000
          TRACE5        =      MASTERBOOLEAN.[ 5:1]#,                      00018100
          TRACE6        =      MASTERBOOLEAN.[ 6:1]#,                      00018200
          TRACE7        =      MASTERBOOLEAN.[ 7:1]#,                      00018300
          TRACE8        =      MASTERBOOLEAN.[ 8:1]#,                      00018400
          TRACE9        =      MASTERBOOLEAN.[ 9:1]#,                      00018500
          IDBLANKS      =      MASTERBOOLEAN.[10:1]#,                      00018600
          FIRSTLINE     =      MASTERBOOLEAN.[11:1]#,                      00018700
          ENTERTOGGLE   =      MASTERBOOLEAN.[12:1]#,                      00018800
          NEWSPACE      =      MASTERBOOLEAN.[13:1]#,                      00018900
          INPASS2       =      MASTERBOOLEAN.[14:1]#,                      00019000
          STRINGTEST    =      MASTERBOOLEAN.[15:1]#,                      00019100
          EOFMARK       =      MASTERBOOLEAN.[16:1]#,                      00019200
          SCANNERROR    =      MASTERBOOLEAN.[17:1]#,                      00019300
          STRINGTYPE    = 4#,                                              00019400
          IDENTTYPE     = 3#,                                              00019500
```

```
        REALTYPE       = 2#,                                                 00019600
        INTEGERTYPE    = 1#,                                                 00019700
        LEFTLINK       = [25:13]#,                                           00019800
        RITELINK       = [12:13]#,                                           00019900
        TABLE[TABLE1]=                                                       00020000
         BIGTAB[(T*ST[:=(TABLE1)).[12:4],                                    00020100
    REAL(BOOLEAN(511) AND BOOLEAN(T*STI))]#,                                 00020200
   TYPEFIELD=[36,5]#,                                                        00020300
    COUNTYPEFIELD=[36:11]#,                                                  00020400
        COUNTFIELD     = [31: 6]#,                                           00020500
        BIGTABSIZE     =  8192  #,                                           00020600
        SOURCELIST     = CONTROLCARDOPTN [0] . [46:1]#;                      00020700
 ALPHA ARRAY                                                                 00020800
        BIGTAB [ 0: BIGTABSIZE DIV 512,0:511 ] ;                             00020900
            BOOLEAN ARRAY  CONTROLCARDOPTN [ 0 : 64 ] ;                      00021000
 ALPHA ARRAY CARDBUF [ 0 : 14 ] ; POINTER CBPOI,SYLPH,SCP,CRP,PT;           00021100
 ALPHA ARRAY                                                                 00021200
        SYMBUF[0:14],  SCRATCH[0:29],                                        00021300
         IDNAME [0:11] ,  POINT [0:12] ,                                     00021400
        RS,COMM[0:100];                                                      00021800
FILE CARD(INTMODE=4,KIND=9,MAXRECSIZE=14,FILETYPE=7);                        00022700
 FILE LINE (MYUSE = 2,KIND = 135,BUFFERS = 3, MAXRECSIZE = 20);              00022800
 FILE STATION (KIND = 3, MYUSE = 3, MAXRECSIZE = 14, INTMODE = 4) ;          00022900
FORMAT FINAL ("NUMBER OF ERRORS DETECTED = ", I4, 4"00"/                     00023000
            "GOOD BYE", 4"00"),                                              00023100
     INVALIDRSREFERENCE("INVALID RS REFERENCE") ;                            00024000
                                                                             00024100
 PROCEDURE NEXTCHAR(SKIPQ) ; VALUE SKIPQ ; BOOLEAN SKIPQ ;   FORWARD ;      00024200
                                              COMMENT AT 51400 ;             00024300
 INTEGER PROCEDURE LASTCHARACTER : FORWARD ;     COMMENT AT 59500 ;          00024400
 INTEGER PROCEDURE ADVANCECHAR : FORWARD ;       COMMENT AT 53500 ;          00024500
PROCEDURE READALINE :  FORWARD :                 COMMENT AT 48200 ;          00024600
PROCEDURE USEPOINT :   FORWARD :                 COMMENT AT 86700 ;          00024700
 REAL PROCEDURE NUMERICLIT ( MANTISSA , DECPT ) ;                            00024900
 VALUE DECPT , MANTISSA : REAL MANTISSA , DECPT ;                            00025000
 FORWARD ;                                       COMMENT AT 46400 ;          00025100
 REAL PROCEDURE INTEGEREAD ( SIGN , DECPT ) ;                                00025200
 VALUE SIGN , DECPT :                                                        00025300
 INTEGER DECPT ; BOOLEAN SIGN : FORWARD ;        COMMENT AT 44100 ;          00025400
 ALPHA PROCEDURE SCAN ; FORWARD ;                COMMENT AT 59800 ;          00025600
 PROCEDURE ALPHAGET ; FORWARD ;                  COMMENT AT 29900 ;          00025700
 PROCEDURE STRINGGET ; FORWARD ;                 COMMENT AT 30900 ;          00025800
                                                                             00025900
 DEFINE SETO(S,E,T,SET4)=REPLACE S+(E) BY SET4 FOR T#;                       00026700
                                                                             00026750
 DEFINE LOOK(LOOK1,LOOK2,LOOK3)= REAL(LOOK1+LOOK2,LOOK3)#;                   00026800
                                                                             00026850
 INTEGER PROCEDURE GOBBLEUPINTEGER(S,D);VALUE S;                             00026900
 POINTER S;INTEGER D;                                                        00027000
 BEGIN                                                                       00027100
      SCAN S FOR Q:8 WHILE GEQ "0";                                          00027200
      D:=INTEGER(S,(GOBBLEUPINTEGER:=8-Q));                                  00027300
 END ;                                                                       00027400
 INTEGER PROCEDURE GOBBLEUPALPHANUMERIC(S,D);                                00027500
     VALUE S,D;POINTER S,D;                                                  00027600
     BEGIN                                                                   00027700
        REPLACE D:D BY S FOR Q:64 WHILE IN ALFANUM[0];                       00027800
        GOBBLEUPALPHANUMERIC:=64-Q;
         REPLACE D BY " " FOR 8;
    END ;

 INTEGER PROCEDURE GOBBLEUPSTRING(S,D);
      VALUE S,D;POINTER S,D;
    BEGIN LABEL L ;  POINTER Q;
         Q:=S;
          WHILE TRUF DO
         BEGIN REPLACE D:D BY Q:Q FOR 1;
                   IF Q=""" OR Q="%" THEN GO L;
        END;
    L:   GOBBLEUPSTRING :=DELTA(S,Q);REPLACE D BY " " FOR 8;
    END ;

 INTEGER PROCEDURE GETNEXTCHAR(S,CHAR);VALUE S;
 POINIER S;INTEGER CHAR;
 BEGIN
        SCAN S:S FOR Q:64 WHILE = " ";
      CHAR:=REAL(S,1);
      GETNEXTCHAR:=64-Q;
 END ;

 PROCEDURF ALPHAGET ;
    BEGIN
         I := 0 ;     NAMEQ :=  IDENTTYPE ;
         DO BEGIN
         I:= (J:= GOBBLEUPALPHANUMERIC(CBPOI+(NCR-1).[6:7],
           SYLPH+I.[6:7])) + I;
         NCR := NCR + J - 1 ; NEXTCHAR(IDBLANKS) ;
         SCANNERROR := I > 63 ; I := I.[ 5:6] ;
         END UNTIL NOT(NXTCHR IN ALFANUM[0]):CHARCOUNT:=I;
    END ;

 PROCEDURE STRINGGET;
    BEGIN
       STRINGTEST:=TRUE;NAMEQ:=STRINGTYPE;I:=0;NEXTCHAR(FALSE);
         DO BEGIN
         I:= (J:= GOBBLEUPSTRING(CBPOI+(NCR-1).[6:7],
               SYLPH + I.[6:7]))+I;
         NCR := NCR + J - 1  ;
         SCANNERROR := I > 63  OR SCANNERROR ; I := I.[ 5:6];
         END UNTIL ADVANCECHAR = """ ;
  CHARCOUNT:=I;
     STRINGTEST := FALSE ; NEXTCHAR(FALSE) ;
 END ;

 INTEGER PROCEDURE COMPARE(S,D,N);
 VALUE S,D,N:INTEGER N;POINTER S,D;
         BEGIN
           IF S<D FOR N THEN COMPARE:=0 ELSE
           IF S=D FOR N THEN COMPARE:=1 ELSE
            COMPARE:=2;
         END;

 DEFINE MOVE(MOVE1,MOVE2,MOVE3)=REPLACE MOVE2 BY MOVE1 FOR MOVE3 WORDS#;
```

```
PROCEDURE TWSTINITIAL ;
 BEGIN
          ENTERTOGGLE := TRUE ;
          LSPACES := 1 ;  BASE := 10 ;  LRS := 0 ;
          ALTERNATIVE := 0 ;
          WRITE (STATION, <"GO AHEAD :",4"00">) ;
          CBPOI := POINTER(CARDBUF) ;  SYLPH := POINTER(SYMBUF) ;
 END TWSTINITIAL ;

COMMENT  TWSTINITIAL CONTAINS THE INITIALIZATION PROCEDURES NECESSARY
         BEFORE READING A NEW STATEMENT.  MOST TWST65 GENERATED
         COMPILERS WORK UNDER THE ASSUMPTION THAT ONLY ONE INPUT STRING
         WILL BE HANDLED.  HOWEVER, OL2/PARSER EXAMINES EACH NEW
         STATEMENT INDEPENDENTLY.  THIS IS DONE TO FACILITATE CONTROL OF
         THE PARSING MECHANISM, PRIMARILY BY FORCING IT TO START OVER
         FOR EACH STATEMENT.  FOR FUTURE MODIFICATIONS, THIS FEATURE
         WILL ENABLE PROCEDURES HANDLING BLOCK LEVELS OR SYMBOL TABLES
         OR WHATEVER TO BE WRITTEN INTO OL2/SKELETON, WHICH IS BETTER
         THAN EMBEDDING THEM IN OL2/TWST AS SEMANTIC ROUTINES, SINCE
         TWST65 ENFORCES A LIMIT ON THE NUMBER OF LINES OF CODE ALLOWED
         IN AN INLINE SEMANTIC ROUTINE, AND SINCE TWST65 ENCODES ANY
         SEMANTIC ROUTINE ENTERED AS A NONTERMINAL (WHICH WOULD BE THE
         ONLY WAY OF CALLING SUCH A ROUTINE FROM MANY PLACES) AS A
         BOOLEAN PROCEDURE.                                        ;


 REAL PROCEDURE INTEGEREAD(SIGN,DECPT) ;
 VALUE SIGN,DECPT ;
 INTEGER DECPT ;  BOOLEAN SIGN ;
      BEGIN
LABEL       LOOP,SKYPP;
     I := J := EXPONENT := MANTISSA := 0 ;
     IF (SIGN := NXTCHR = "-") OR NXTCHR = "+" THEN NEXTCHAR(FALSE) ;
LOOP:
                IF NXTCHR = "." THEN DECPT:=DECPT+1 ELSE
                     BEGIN
                SCRATCH[0]:=0;
                REPLACE POINTER(SCRATCH)+5 BY NXTCHR FOR 1 WITH
                 WEIGHTS[0];
             IF SCRATCH[0] GEQ BASE THEN GO TO SKYPP;
                     MANTISSA:= MANTISSA * BASE +SCRATCH[0];
                     EXPONENT := EXPONENT + DECPT ;
         END;
                     NEXTCHAR(FALSE) ;
           GO TO LOOP;
SKYPP:
     INTEGEREAD := IF (SIGN) THEN -MANTISSA ELSE MANTISSA ;
     SCANNERROR := DECPT > 1 ;
     END ;

 REAL PROCEDURE NUMERICLIT(MANTISSA,DECPT) ;
 VALUE DECPT,MANTISSA ; REAL MANTISSA,DECPT ;
 BEGIN
    MANTISSA := IF NXTCHR = "@" THEN 1 ELSE INTEGEREAD(FALSE,DECPT);
    IF NXTCHR = "@" THEN
          BEGIN
```

```
               NEXTCHAR(FALSE) ;
               I := EXPONENT - INTEGEREAD(FALSE,0);
               SCANNERROR := SCANNERROR OR EXPONENT  NEQ 0 ;
               EXPONENT := I ;
          END ;
     IF EXPONENT = 0  THEN
         BEGIN NUMERICLIT:= I:= MANTISSA ; NAMEQ:= INTEGERTYPE END ELSE
         BEGIN NUMERICLIT := MANTISSA * BASE **(-EXPONENT) ;
              NAMEQ := REALTYPE END ;
     CHARCOUNT := 3 ;
 END ;

PROCEDURE READALINE ;
    BEGIN
         READ (STATION, 13, CARDBUF[*]) ;
         REPLACE CRPOI + LASTCOL + 1 BY "%" ;
         NCR := FIRSTCOL ;
         IF NOT USEPT THEN WRITE (LINE, 13, POINT[*]) ;
         WRITE (LINE, 13, CARDBUF[*]) ;
         USEPT := TRUE ;
    END READALINE ;


COMMENT  READALINE IS INVOKED WHENEVER THE OL2/PARSER NEEDS INPUT,
         WHETHER IN THE FORM OF A NEW STATEMENT OR A CONTINUATION OF A
         PREVIOUS STATEMENT.  NEW STATEMENTS ARE CALLED FOR FROM
         ENDOFPROGRAMDEFINE- CONTINUATIONS ARE CALLED FOR FROM NEXTCHAR.

         READALINE ALSO PRINTS OUT THE ERROR POINTER (CONTAINED IN THE
         ALPHA ARRAY POINT, ESTABLISHED BY USEPOINT) FOR THE PREVIOUS
         STRING OF CHARACTERS, IF ANY ERROR WAS DETECTED.  READALINE
         THEN ALWAYS PRINTS THE INPUT STRING FOR HARDCOPY FOR THE USER.

         NEXTCHAR, FOLLOWING, HAS BEEN MODIFIED TO DISPLAY THE MESSAGE
         'CONTINUE :' AND CALL READALINE WHEN A SYNTACTICALLY CORRECT
         PARTIAL STRING HAS BEEN PARSED AND A CONTINUATION IS NECESSARY.
         THE LOGIC WHICH DETERMINES WHEN TO SEEK MORE DATA IS UNCHANGED
         FROM TWST65/SKELETON.                                         ;


 PROCEDURE NEXTCHAR ( SKIPQ ) ; VALUE SKIPQ ; BOOLEAN SKIPQ ;
      IF STRINGTEST THEN
        BEGIN
      IF (NCR := NCR + 1) > LASTCOL THEN BEGIN
             WRITE (STATION,<"CONTINUE :",4"00">);  READALINE ; END ;
        IF EOFMARK THEN NCR:=1;
        END ELSE
      IF SKIPQ THEN
             DO
             BEGIN
    IF NXTCHR = "%" THEN BEGIN  WRITE (STATION,<"CONTINUE :",4"00">);
                     READALINE ;  END   ELSE NCR := NCR + 1 ;
             IF EOFMARK THEN NCR:=1          ELSE
             NCR := NCR + GETNEXTCHAR(CRPOI+(NCR-1),TWSTI ) %
             END UNTIL NXTCHR  NEQ "%" AND NXTCHR  NEQ " "
        ELSE BEGIN
             DO
```

```
BEGIN
     IF NXTCHR = "%" THEN BEGIN  WRITE (STATION,<"CONTINUE :",4"00">));
                           READALINE ;  END  ELSE NCR := NCR + 1 ;
END
              UNTIL NXTCHR NEQ "%" AND NXTCHR NEQ " "
         OR NXTCHR= " " AND NOT SKIPQ;%
         END ;

 INTEGER PROCEDURE ADVANCECHAR ;
 BEGIN NEXTCHAR(FALSE) ; ADVANCECHAR := REAL(NXTCHR,1)  END ;

 INTEGER PROCEDURE TABLESEARCH ( STRING,COUNT,TYPE) ;
 VALUE COUNT,TYPE ;
 INTEGER COUNT , TYPE ;
 ARRAY STRING[0] ;
 COMMENT SEARCH BIGTAB FOR THE STRING OF COUNT CHARACTERS
         OF TYPE TYPE ;
 BEGIN COMMENT IN THIS PROCEDURE THE VARIABLES MANTISSA AND EXPONENT
               ARE USED AS SCRATCH  VARIABLES ;
             COMMENT COMPUTE A HASH ADDRESS FOR THE ITEM IN BIGTAB;
             COMMENT COMPUTE A HASH ADDRESS FOR THE ITEM IN BIGTAB;
MANTISSA:=ENTIER(ABS((0&STRING[0][3:43:4]&STRING[0][7:35:4]
&STRING[0][11:27:4]&STRING[0][15:19:4]&STRING[0][19:11:4]&
STRING[0][23:3:4]) MOD 250));
             IF BIGTAB[0,MANTISSA]=0 THEN
                       BEGIN
     IF ENTERTOGGLE THEN BIGTAB[0,MANTISSA] := BTABPTR ;
                      EXPONENT := 1 ;  I := MANTISSA ;   J := 0 ;
                      END ELSE
             BEGIN
                 K := BIGTAB[0,MANTISSA] ;   EXPONENT := 1 ;
                 DO BEGIN
                 I := K ; J := TABLE[ I ] . COUNTFIELD ;
                 IF MANTISSA:=SIGN(J-COUNT)+1=1 THEN
                 MANTISSA:=COMPARE(POINTER(STRING[0]),
                 POINTER(TABLE[I+1]),MIN(J,COUNT));
                 IF MANTISSA = 2 OR  MANTISSA = 1 AND COUNT > J
                     THEN BEGIN J := 0 ;
                                K := TABLE[I] . RITELINK END ELSE
                 IF MANTISSA = 0 OR  MANTISSA = 1 AND COUNT < J
                     THEN BEGIN J := 1 ;
                                K := TABLE[I] . LEFTLINK END ELSE
                   EXPONENT := 0 ;
             END UNTIL NOT BOOLEAN(EXPONENT) OR K = 0 ;
             END ;
             COMMENT NO I CONTAINS THE LAST LINKED ENTRY IN
                   THE TABLE - IF EXPONENT IS SET THEN NO ENTRY
                   NEED BE MADE SINCE THE ITEM IS ALREADY
                   PRESENT. IF J IS ONE THEN THE LAST LINK WAS  LEFT :
             IF BOOLEAN(EXPONENT)  THEN
             IF ENTERTOGGLE        THEN
                 BEGIN
                 IF BTABPTR.[ 8:9] + COUNT DIV 6   GEQ 511 THEN
                      COMMENT NOT ENOUGH ROOM IN THIS RUN ;
                      BTABPTR := 0 & BTABPTR[12:12:4] + 512  ;
                 MOVE(POINTER(STRING),POINTER(TABLE[BTABPTR+1]), %
                     (COUNT+5)DIV 6)          ;
```

```
        TABLE[BTABPTR].COUNTYPEFIELD:=(COUNT&TYPE[10:4:5]);          00058300
              K := BTABPTR ;                                          00058400
              IF BOOLEAN(J) THEN TABLE[ I ] . LEFTLINK := BTABPTR    00058500
                           ELSE TABLE[ I ] . RITELINK := BTABPTR ;00058600
              IF BTABPTR := BTABPTR + COUNT DIV  6 + 1 +              00058700
                  REAL(COUNT MOD  6  NEQ 0) > BIGTABSIZE              00058800
                 THEN TYPE := 1/(TYPE:=0);                            00058900
              I := K ;                                                00059000
              END ELSE   I := 0 ;                                     00059100
             TABLESEARCH:=I; %                                        00059200
 IF I NEQ 0 AND (TYPE=IDENTTYPE ) THEN NAMEQ:=I;                      00059300
END TABLESEARCH ;                                                     00059400
                                                                      00059450
INTEGER PROCEDURE LASTCHARACTER ;                                     00059500
COMMENT RETURNS THE CURRENT VALUE OF NXTCHR THEN ADVANCES THE READER ; 00059600
BEGIN LASTCHARACTER:=NAMEQ:=J:=REAL(NXTCHR,1); NEXTCHAR(FALSE); END;  00059700
                                                                      00059750
ALPHA PROCEDURE SCAR ;                                                00059800
    BEGIN                                                             00059900
    COMMENT THE GLOBAL VARIABLE NAME CONVEYS THE TYPE OF THE          00060000
            ITEM READ. IT IS SET IN EACH OF THE PROCEDURES            00060100
            ALPHAGET,STRINGET,NUMERICLIT AND HAS THE FOLLWOING VALUES 00060200
            NAME = 3          IDENTIFIER,                             00060300
            NAME = 4          STRING,                                 00060400
            NAME = 1          INTEGER,                                00060500
            NAME = 2          REAL ;                                  00060600
   COMMENT SCANMODE = 0     NORMAL OPERATING MODE: ASSEMBLES          00060700
                            IDENTIFIERS, NUMBERS AND STRINGS          00060800
                            FOLLOWED BY TABLE LOOK UP.                00060900
           SCANMODE = 4     SAME AS 0 EXCEPT TABLE LOOK UP IS         00061000
                            SUPRESSED. SCAR RETURNS THE SIX CHARACTER 00061100
                            BCD OF IDENTIFIERS AND STRINGS            00061200
                            AND THE VALUE OF NUMBERS .                00061300
          SCANMODE = 1      SAME AS 5 EXCEPT THAT MULTIPLE BLANKS     00061400
                 ARE COMPRESSED TO SINGLE BLANKS .                    00061500
          SCANMODE = 5     RETURNS EVERY CHARACTER ;                  00061600
   IF SCANMODE = 0 OR SCANMODE = 4 THEN                               00062600
       BEGIN                                                          00062700
       NAMEQ := 0;SYMBUF[0]:="      ";                                00062800
      IF NXTCHR = " " THEN NEXTCHAR ( TRUE ) ;                        00062900
       IF  NXTCHR IN LOOKAGAIN[0] THEN                                00063000
           SCAR:= LASTCHARACTER ELSE                                  00063100
       IF NXTCHR IN ALFA[0]    THEN ALPHAGET ELSE                     00063200
     BEGIN                                                            00063300
         SCRATCH[0]:=0;                                               00063400
         REPLACE POINTER(SCRATCH)+5 BY NXTCHR FOR 1 WITH              00063500
          WEIGHTS[0];                                                 00063600
      IF SCRATCH[0]<BASE THEN                                         00063700
             SYMBUF[0] := NUMERICLIT(0,0) ELSE                        00063800
       IF NXTCHR = "." THEN                                           00063900
     BEGIN                                                            00064000
         TWSTI:=ADVANCECHAR;                                          00064100
         SCRATCH[0]:=0;                                               00064200
         REPLACE POINTER(SCRATCH)+5 BY NXTCHR FOR 1 WITH              00064300
          WEIGHTS[0];                                                 00064400
            IF SCRATCH[0]  < BASE THEN                                00064500
               SYMBUF[0] := NUMERICLIT(0,1) ELSE                      00064600
         NAMEQ:= J:= SCAR:= ".", END  ELSE                                 00064700
    IF NXTCHR = "@" THEN                                                   00064800
  BEGIN                                                                    00064900
       TWSTI:=ADVANCECHAR;                                                 00065000
       SCRATCH[0]:=0;                                                      00065100
       REPLACE POINTER(SCRATCH)+5 BY NXTCHR FOR 1 WITH                     00065200
        WEIGHTS[0];                                                        00065300
          IF NXTCHR        = "+" OR NXTCHR = "-" OR                        00065400
            SCRATCH[0]                                 LEQ BASE THEN       00065500
               BEGIN                                                       00065600
               SYMBUF[0] := BASE ** INTEGEREAD (FALSE,0) ;                 00065700
               CHARCOUNT := 6 ; NAMEQ:=REALTYPE END ELSE                   00065800
          NAMEQ:= J:= SCAR:= "@"; END ELSE                                 00065900
   IF NXTCHR = """ THEN STRINGGET ELSE                                     00066000
         SCAR:= LASTCHARACTER ;                                            00066100
  END;                                                                     00066200
    IF NAMEQ < 10 THEN                                                     00066300
                       COMMENT OTHER THAN SIMPLE CHARACTER ;               00066400
           SCAR:=J:= IF SCANMODE  = 4 THEN  SYMBUF[0]                      00066500
              ELSE                                                         00066600
                       TABLESEARCH( SYMBUF , CHARCOUNT , NAMEQ ) ;         00066700
    END ELSE                                                               00066800
    IF SCANMODE = 5 THEN SCAR:= LASTCHARACTER ELSE                         00066900
    IF SCANMODE = 1 THEN                                                   00067000
         DO SCAR:=LASTCHARACTER UNTIL NXTCHR  NEQ " "       ELSE           00067100
       SCANNERROR:= TRUE ;                                                 00067200
 END SCAR ;                                                                00067700
                                                                           00067750
DEFINE MSTACK = COMM#,TETTL=SEMTM# ;                                       00067800
DEFINE LEFTPREC                = [29:6]#,                                  00074600
       RITEPREC                = [23:6]#,                                  00074700
       WHICHPREC               = [35:6]#,                                  00074800
       WHICHEXEC               = [41:6]#,                                  00074900
       WORDLINK                = [12:13]# ,                                00075000
       LISTCOUNTFIELD          = [45:9]#, %                                00075100
       LISTCOUNT(LISTCOUNT1) = RS[LISTCOUNT1].LISTCOUNTFIELD#,             00075200
       LASTFETCH            = NAMEQ#,                                      00075300
       SEMANTICS(SEMANTICS1,SEMANTICS2,SEMANTICS3)=                        00075400
             BEGIN FIRST := SEMANTICS2 : LAST:= SEMANTICS3 ;               00075500
             EXEC(SEMANTICS1) END#,                                        00075600
          PM9 = COMM[FIRST -10]#,PM8 = COMM[FIRST - 9]#,                   00075700
          PM7 = COMM[FIRST - 8]#,PM6 = COMM[FIRST - 7]#,                   00075800
          PM5 = COMM[FIRST - 6]#,PM4 = COMM[FIRST - 5]#,                   00075900
          PM3 = COMM[FIRST - 4]#,PM2 = COMM[FIRST - 3]#,                   00076000
          PM1 = COMM[FIRST - 2]#,P0  = COMM[FIRST - 1]#,                   00076100
          P1  = COMM[FIRST     ]#,P2  = COMM[FIRST + 1]#,                  00076200
          P3  = COMM[FIRST + 2]#,P4  = COMM[FIRST + 3]#,                   00076300
          P5  = COMM[FIRST + 4]#,P6  = COMM[FIRST + 5]#,                   00076400
          P7  = COMM[FIRST + 6]#,P8  = COMM[FIRST + 7]#,                   00076500
          P9  = COMM[FIRST + 8]#,P10 = COMM[FIRST + 9]#,                   00076600
          P11 = COMM[FIRST +10]#,P12 = COMM[FIRST +11]#,                   00076700
          P13 = COMM[FIRST +12]#,P14 = COMM[FIRST +13]#,                   00076800
          P15 = COMM[FIRST +14]#,P16 = COMM[FIRST +15]#,                   00076900
          P17 = COMM[FIRST +16]#,P18 = COMM[FIRST +17]#,                   00077000
          P19 = COMM[FIRST +18]#,P20 = COMM[FIRST +19]#,                   00077100
          P21 = COMM[FIRST +20]#,P22 = COMM[FIRST +21]#,                   00077200
          P23 = COMM[FIRST +22]#,P24 = COMM[FIRST +23]#,                   00077300
```

```
               P25 = COMM[FIRST +24]#,P26 = COMM[FIRST +25]#,
               P27 = COMM[FIRST +26]#,P28 = COMM[FIRST +27]#,
               P29 = COMM[FIRST +28]#,P30 = COMM[FIRST +29]#,
               P31 = COMM[FIRST +30]#,P32 = COMM[FIRST +31]#,
               P33 = COMM[FIRST +32]#,P34 = COMM[FIRST +33]#,
               P35 = COMM[FIRST +34]#,P36 = COMM[FIRST +35]#,
               P37 = COMM[FIRST +36]#,P38 = COMM[FIRST +37]#,
               P39 = COMM[FIRST +38]#,P40 = COMM[FIRST +39]#,
               PLAST = COMM[LAST]#;

PROCEDURE GAP (N) ;
    VALUE N ;
    INTEGER N ;
    BEGIN
    FOR TWSTI:=LRS STEP -1 UNTIL N    %
    DO
       BEGIN
        RS  [TWSTI+1]:= RS[TWSTI]; %
        COMM[TWSTI+1]:= COMM[TWSTI]; %
        END ;
   LRS:= LRS + 1 ;
   CUMM[N]:= RS[N]:= 0;
   END ;

PRUCEDURE DELETE (M, N) ; VALUE M,N; INTEGER M,N; %
    BEGIN IF N=0 THEN BEGIN GAP(M);RS[M]:=-1;END ELSE %
          BEGIN IF M:=M+N  LEQ LRS AND N:=N-1 > 0    %
          THEN
                FOR TWSTI:=M STEP 1 UNTIL LRS DO %
                BEGIN  RS[TWSTI-N]:=RS[TWSTI]; %
                       COMM[TWSTI-N]:=COMM[TWSTI]; %
                END; %
                LRS:=LRS-N; %
         END;END; %

REAL PROCEDURE FETCH(N)); %
    VALUE NO ;
    INTEGER NO ;
         BEGIN
              IF NO > LRS THEN
              BEGIN
                   COMM [ LRS := LRS + 1 ] := SCAR ;
                   FETCH := RS [LRS] := NAMEQ;
              IF NO > LRS THEN
                  BEGIN WRITE(LINE,INVALIDRSREFERENCE) ;
                        NO := 1/(NO:=0)
                  END ;
              END
     ELSE IF NO=LRS THEN FETCH:=LASTFETCH:=RS[NO]
              ELSE WHILE FETCH:=LASTFETCH:=RS[NO]=-1 DO DELETE(NO-1,2);
         END ;

RnCEDURE USEPOINT ;
   BEGIN
     IF USEPT THEN BEGIN
        USEPT := FALSE ;
        PT := POINTER(POINT[0]) ;
```

```
          REPLACE PT BY 4"00" FOR 76 ;
          REPLACE PT BY " " FOR NCR - 1 ;
          PT := PT + NCR - 1 ;
          REPLACE PT BY "<" ;
          WRITE (STATION, 13, POINT[*]) ;
          NUMERRS := NUMERRS + 1 ;
          END ;
          SEMANTICTEST := FALSE ;
     END USEPOINT ;

COMMENT  THE PROCEDURE USEPOINT DISPLAYS A "<" UNDER THE FIRST CHARACTER
         NOT SCANNED WHEN THE FIRST ERROR IS FOUND IN AN OL/2 STATEMENT.
         SINCE THE ERROR ROUTINES ALL FORCE THE PARSER TO EXAMINE THE
         NEXT ALTERNATIVE, IN EFFECT FORCING THE PARSER TO RETRACE ITS
         PATH DOWN THE PARSING TREE, THE BOOLEAN VARIABLE USEPT IS SET
         TO FALSE SO THAT ONLY ONE "<" WILL BE DISPLAYED UNDER AN
         INCORRECT OL/2 STATEMENT.                    ;


DEFINE ENDOFPROGRAMDEFINE =
   BEGIN
     REPLACE POINTER(IDNAME[0]) BY "WARNING : " ;
     REPLACE POINTER(IDNAME[0]) BY "HASN'T BEEN PPEVIOUSLY DECLARED" ;
     REPLACE POINTER(POINT[0]) BY " " FOR 78 ;
     WRITE (LINE, < X15, "***** OL2/PARSER ***** OL2/PARSER ***** OL2/PA
RSER ***** OL2/PARSER *****              " //// "ALL STATEMENTS ENTERED AR
E LISTED ** INCORRECT STATEMENTS ARE FLAGGED UNDER THE FIRST CHARACTER N
OT SCANNED WITH A ",4"(04C70">) ;

     MASTERBOOLEAN := BOOLEAN(NUMERRS) ;
     FIRSTCOL := 1 ;  LASTCOL := 74 ;  FINISH := FALSE ;
     NUMERRS := 0 ;
     WRITE (STATION, <"REMEMBER TO TYPE 'FINISH' TO TERMINATE OL2/PARSER
">) ;
     WHILE NOT FINISH DO BEGIN
          TWSTINITIAL ;
          READALINE ;
          IF CALL THEN WRITE (STATION,<"PARSE COMPLETE",4"00">)
                ELSE WRITE (STATION,<"PARSE INCOMPLETE",4"00">) ;
     END ;
     WRITE (STATION, <"A LISTING OF YOUR STATEMENTS IS BEING PRINTED",
4"00"/"PICK IT UP AT THE ROUTING WINDOW",4"00">) ;
     WRITE (STATION, FINAL, NUMERRS) ;
     END ;  END ;  %;

COMMENT  ENDOFPROGRAMDEFINE IS INSERTED BY TWST65 BEFORE THE FINAL 'END'
         IN THE ALGOL SOURCE CODE FOR OL2/PARSER.  EXCEPT FOR THE
         INSTRUCTIONS SETTING THE SIZE OF BIGTAB AND THE LOCATION OF
         THE LAST RESERVED WORD IN BIGTAB AND THOSE FILLING BIGTAB, THE
         INSTRUCTION 'ENDOFPROGRAMDEFINE' IS THE FIRST EXECUTABLE
         INSTRUCTION IN OL2/PARSER.

         OL2/PARSER CONSISTS OF A SERIES OF PROCEDURE CALLS.
         ENDOFPROGRAMDEFINE CONTAINS THE INITIALIZATION NECESSARY FOR
         THE ENTIRE PROGRAM, SUCH AS SETTING UP THE ALPHA ARRAY POINT
         AND THE PRINTING OF HEADINGS, AND THE LOGIC NECESSARY TO
         CONTINUE READING STATEMENTS AND INVOKING THE PARSING ROUTINES
```

```
        UNTIL THE WORD 'FINISH' IS ENTERED AS INPUT.                        00092000
                                                                            00093000
        'CALL' IS DEFINED BY T ST65 TO BE THE BOOLEAN PROCEDURE             00093100
        'TESTSTATEMENT', AND THEREFORE WILL INITIATE THE PARSE FOR          00093200
        STATEMENT AS DEFINED IN DL2/TWST. THE SOURCE FILE FOR               00093300
        TWST65 IN THE GENERATION OF DL2/PARSER.  IF CALL = TR   UPON        00093400
        RETURN, THEN THE PARSER HAS REACHED A TERMINALNODE IN THE           00093500
        PARSING STRUCTURE.  IF CALL = FALSE UPON RETURN, THEN THE           00093600
        PARSE HAS FAILED WITHIN THE TREE, AND THE INPUT STRING HAS          00093700
        NOT BEEN COMPLETELY EXAMINED.              ;                        00093800
                                                                            00093900
                                                                            00094000
BEGIN           COMMENT  SECOND UNMATCHED BEGIN ;                           00094100
```

APPENDIX D

FILE MAINTENANCE

The files OL2/SYNTAX, OL2/TWST, OL2/SKELETON, OL2/PARSER, and OL2/INFORMATION are stored on small tape number 210, named OL2SYNTAX, which currently is kept in the B6500 machine room in Coordinated Science Laboratory. All of these files except OL2/PARSER may be modified using the file editing features implemented on the B6500. The file editor and its use are described in Able (5).

Card versions of OL2/SYNTAX, OL2/TWST, OL2/SKELETON, and OL2/INFORMATION are available from Professor J. R. Phillips of the Department of Computer Science.

The current version of OL2/PARSER was generated by the TWST65 compiler-compiler on the B6500 from the files OL2/TWST (Appendix B) and OL2/SKELETON (Appendix C), which are maintained as noted above.

It is advisable to place the card files on disk before regenerating OL2/PARSER; tape files must be placed on disk before they can be used.

To establish a disk file from cards, run the following job. The '?' character indicates an illegal punch; an illegal punch in column one denotes a control card.

```
?USER = <some valid user code>
?EXECUTE CARD/DISK
?FILE DISK = OL2/TWST      (or other file name)
?DATA
<Appropriate card deck>
?END
```

To load the tape files to disk, first ask an operator to mount small tape number 210 without a ring. Then, from a terminal (see Appendix E for information about Hazeltines), enter the following commands.

```
-SHIFT-N-LIBMAIN
COPY  OL2/TWST, OL2/SKELETON FROM OL2SYNTAX   (or other file names)
END
```

Once the files are on disk, they may be modified using the editor as described in Abel (5).

OL2/PARSER is generated in two passes, one through TWST65, the other through the ALGOL compiler.

First, run TWST65.

```
?EXECUTE TWST65/COMPILE
?FILE SKELETON = OL2/SKELETON DISK SERIAL
?FILE CARD = OL2/TWST DISK SERIAL
?END
```

After the previous job has run to completion, run the ALGOL compiler.

```
?COMPILE OL2/PARSER WITH ALGOL LIBRARY
?ALGOL FILE CARD = OL2/SOURCE DISK SERIAL
?END
```

It would be best at this point to have small tape number 210 mounted with a ring and to store the new versions of OL2/TWST, OL2/SKELETON, and OL2/PARSER on the tape. See Abel (5) for details on using the library maintenance routine. Previous versions may be kept for backup.

APPENDIX E

OL2/INFORMATION

**** WHAT YOU NEED TO KNOW TO USE THE OL/2 INTERACTIVE PARSER ****

THIS DOCUMENT HAS THREE SECTIONS --
    SECTION 1 DESCRIBES THE HAZELTINE TERMINALS AVAILABLE IN THE B6500 MACHINE ROOM.
    SECTION 2 DETAILS THE PROCEDURE NECESSARY TO LOAD "OL2/PARSER" INTO THE B6500 SYSTEM.
    SECTION 3 EXPLAINS HOW TO USE "OL2/PARSER".

***** SECTION 1 *****

THERE ARE USUALLY FIVE OR SIX HAZELTINE CRT TERMINALS OPERATING IN THE B6500 MACHINE ROOM. THE TERMINALS TRANSMIT AND RECEIVE DATA IN THREE MODES, CONTROLLED BY A SWITCH ON THE REAR OF THE DEVICE LABELED - FULL - HALF - HALF BUFF - . THE FEATURES OF THE -HALF BUFF- AND -HALF- MODES WILL BE DESCRIBED.

IN THE -HALF BUFF- MODE, CHARACTERS ENTERED AT THE TERMINAL ARE DISPLAYED AT A HIGHER INTENSITY THAN CHARACTERS WRITTEN BY THE SYSTEM AND ARE CALLED FOREGROUND CHARACTERS. TRANSMISSION OF AN INPUT STRING IS ACCOMPLISHED BY DEPRESSING THE -SHIFT- AND -XMIT- KEYS SIMULTANEOUSLY. A SOLID RECTANGLE IS DISPLAYED AT THE SCREEN POSITION FROM WHICH TRANSMISSION WAS EFFECTED, AND A CARRIAGE RETURN AND LINE FEED FOLLOW THE TRANSMISSION.

IF THERE IS A CHARACTER IN THE LAST (74) POSITION OF THE HAZELTINE LINE, ANY SUBSEQUENT ENTRY WILL OVERWRITE THAT CHARACTER. YOU WILL BE UNABLE TO GET A CARRIAGE RETURN AND LINE FEED UNTIL TRANSMISSION IS EFFECTED, WHICH WILL OVERWRITE THE CHARACTER IN POSITION 74.

ALSO, IN THIS MODE THE BACKSPACE (CHARACTER DELETE = -SHIFT-O-) CAUSES A LEFT-POINTING ARROW TO BE DISPLAYED NEXT TO THE INCORRECT CHARACTER, AND THE CORRECT CHARACTER IS DISPLAYED NEXT TO THE ARROW. THE NET RESULT IS THAT THREE CHARACTERS ARE REQUIRED TO OBTAIN ONE CORRECT ENTRY.

IN THE -HALF- MODE (HALF DUPLEX), USER ENTERED CHARACTERS ARE DISPLAYED AT THE SAME INTENSITY AS THOSE WRITTEN BY THE SYSTEM. THE BACKSPACE (-SHIFT-O-) CAUSES THE CURSOR ON THE SCREEN TO BACK UP TO THE INCORRECT ENTRY SO THAT IT CAN BE OVERWRITTEN. TYPING ERRORS THEREFORE DO NOT IMPAIR SCREEN LEGIBILITY.

TRANSMISSION OF AN INPUT STRING IS ACHIEVED BY HITTING -CR-. HOWEVER, IN THIS MODE, AFTER A CHARACTER IS ENTERED AT POSITION 74, THE TERMINAL EMITS A BEEP AND ADVANCES THE CURSOR TO THE FIRST POSITION OF THE NEXT LINE. "OL2/PARSER" WILL NOT ACCEPT INPUT FROM THE NEW LINE, HOWEVER, IF YOU THEN HIT -CR-, "OL2/PARSER" WILL RECEIVE A FULL 74 CHARACTERS OF INPUT. IF POSITION 74 OCCURS IN THE MIDDLE OF A KEYWORD OR ALPHABETIC OR NUMERIC STRING, YOU NEED NOT DELETE THE ENTIRE LINE (ACCOMPLISHED BY -CTRL-D-). INSTEAD, HIT THE BACKSPACE ENOUGH TIMES TO REMOVE THE PARTIAL STRING.

```
I RECOMMEND USING THE -HALF- MODE WHEN RUNNING "OL2/PARSER" BECAUSE THE
BACKSPACING FEATURE IMPROVES LEGIBILITY AND ALLOWS THE MARKER DISPLAYED
BY "OL2/PARSER" UNDER INCORRECT STATEMENTS TO BE UNDER THE FIRST
CHARACTER NOT SCANNED.


THE HAZELTINE SPECIAL CHARACTERS YOU NEED TO KNOW ARE :

-SHIFT-O-   BACKSPACE OR DELETE CHARACTER
            USED TO CORRECT TYPING ERRORS WHICH ARE QUICKLY NOTICED.

-SHIFT-N-   DISPLAYS AS AN UPWARD-POINTING ARROW.
            WHEN PRESENT IN POSITION 1 OF A LINE, IT IS INTERPRETED BY
            THE DATA COMMUNICATION SYSTEM AS INITIATING A COMMAND TO THE
            B6500 MONITOR.  OTHERWISE, IT IS EQUIVALENT TO THE "NOT"
            CHARACTER IN THE EBCDIC SET ( ¬ = 5F ).

-SHIFT-1-   EXCLAMATION MARK
            IT IS EQUIVALENT TO THE EBCDIC CHARACTER '|' TO "OL2/PARSER".

-CTRL-D-    LINE DELETE
            USED TO DELETE AN ENTIRE LINE WHEN AN ERROR IS MADE.  THIS
            CHARACTER MUST BE TRANSMITTED IN -HALF BUFF- MODE.

-CTRL-E-    WRU
            ASKS THE DATA COMMUNICATION SYSTEM TO IDENTIFY ITSELF.  IT IS
            A GOOD WAY TO FIND WHETHER DATA-COM IS UP.

-CR-        TRANSMIT FOR -HALF- MODE  ( CARRIAGE RETURN ).

-SHIFT-XMIT-  TRANSMIT FOR -HALF BUFF- MODE.


FOR MORE INFORMATION ABOUT THE HAZELTINES AND THE B6500 DATA
COMMUNICATION SYSTEM, CONSULT "THE LITTLE GOLDEN BOOK OF THE B6500",
AVAILABLE THROUGH ILLIAC IV DOCUMENTATION, OR A LISTING OF THE FILE
"EDITOR/DOCUMENT", AVAILABLE THROUGH THE ROUTING WINDOW IN THE B6500
ROOM.



             ***** SECTION 2 *****


SINCE "OL2/PARSER" IS NOT A SYSTEM PROGRAM ON THE B6500, IT IS THE
RESPONSIBILITY OF THE USER TO LOAD THE PROGRAM INTO THE SYSTEM SO IT
MAY BE UTILIZED.

"OL2/PARSER" IS STORED ON SMALL TAPE NO. 210, CALLED "OL2SYNTAX".  THIS
TAPE MUST BE MOUNTED AND "OL2/PARSER" MUST BE COPIED FROM TAPE TO DISK
BEFORE THE PROGRAM CAN BE RUN.

IF YOU ARE RUNNING IN THE B6500 MACHINE ROOM, GO TO THE ROUTING WINDOW
AND ASK AN OPERATOR TO MOUNT SMALL TAPE NO. 210 WITHOUT A RING.  IT IS
A NINE TRACK TAPE.
```

IF YOU ARE NOT RUNNING IN THE B6500 MACHINE ROOM, IT WOULD BE BEST TO CALL 333-7188 AND ASK AN OPERATOR TO MOUNT SMALL TAPE NO. 210 WITHOUT A RING BEFORE YOU DIAL YOUR TERMINAL INTO THE SYSTEM. YOU CAN ALSO BROADCAST A MESSAGE OVER DATACOM TO ASK THE OPERATOR TO MOUNT THE TAPE. BUT THERE IS NO GUARANTEE THAT ANYONE WILL SEE THE MESSAGE.

IN THE MACHINE ROOM, THE HAZELTINES ARE USUALLY LEFT ON, SO ALL YOU NEED DO IS FIND A FREE TERMINAL. IF IT IS NOT ON AND YOU ARE NOT FAMILIAR HOW TO DIAL IT INTO THE SYSTEM OR TURN IT ON, ASK SOMEONE FOR HELP. OTHERWISE, YOU CAN CONNECT YOUR TERMINAL TO THE B6500 BY DIALING 333-808X, WHERE X = 0,1,...,8. BE SURE TO SET YOUR TERMINAL FOR A TRANSMISSION RATE OF 300 BAUD.

HIT -CTRL-E- TO DETERMINE WHETHER DATACOM IS UP. IF IT IS, IT WILL RESPOND WITH A MESSAGE.

IF YOU NOW CHOOSE TO SEND A MESSAGE TO ASK THE OPERATOR TO MOUNT "OL2SYNTAX", TRY SOMETHING SIMILAR TO

```
-SHIFT-N- TO ALL  OPERATOR PLEASE MOUNT SMALL TAPE 210 NO RING
```

THE INITIAL WORDS "TO ALL" ARE ESSENTIAL SINCE THERE IS NO WAY AT THIS TIME TO SEND A MESSAGE ONLY TO THE OPERATOR.

THE SEQUENCE OF INSTRUCTIONS NECESSARY TO LOAD "OL2/PARSER" IS :

```
-SHIFT-N- LIBMAIN    %THIS CALLS THE LIBRARY MAINTAINANCE ROUTINE, WHICH
                      DISPLAYS A COLON WHEN READY TO ACCEPT A COMMAND.

USER = CACPHILLIPS   %USER CODES ARE NOW NECESSARY ON THE B6500.

TAPEDIR OL2SYNTAX    %THIS ASKS FOR THE DIRECTORY OF "OL2SYNTAX".
                      IT IS A GOOD WAY TO FIND WHETHER THE TAPE HAS
                      BEEN MOUNTED YET.

COPY OL2/PARSER FROM OL2SYNTAX
                     %DON'T USE THIS COMMAND UNTIL THE TAPE IS MOUNTED.
                      IF YOU NEED A LISTING OF "OL2/SYNTAX" FOR
                      REFERENCE PURPOSES WHILE RUNNING "OL2/PARSER"
                      USE
                        COPY OL2/PARSER,OL2/SYNTAX FROM OL2SYNTAX
                      INSTEAD.

END                  %THIS WILL TERMINATE LIBRARY MAINTAINANCE.
```

NOW "OL2/PARSER" IS IN THE SYSTEM AND READY TO RUN. IF YOU DESIRE A LISTING OF "OL2/SYNTAX" TYPE

```
-SHIFT-N- RUN DISK/PRINT; FILE DISK = OL2/SYNTAX; END
```

PRIOR TO RUNNING "OL2/PARSER".

***** SECTION 3 *****

THE FOLLOWING COMMANDS WILL INITIATE "OL2/PARSER" WITH YOUR TERMINAL AS INPUT AND OUTPUT.

```
-SHIFT-N- CC
USER = CACPHILLIPS    %ONCE REMOTE CONTROL DISPLAYS A COLON.
RUN OL2/PARSER; END
```

"OL2/PARSER" WILL DISPLAY

```
GO AHEAD :
```

WHEN IT EXPECTS A NEW STATEMENT. TYPE YOUR STATEMENT, OR UP TO 74 CHARACTERS OF THE STATEMENT, FOLLOWED BY THE APPROPRIATE TRANSMISSION KEY (IF NECESSARY -- SEE SECTION 1). NUMERALS, IDENTIFIERS, AND KEYWORDS CANNOT BE CONTINUED FROM ONE LINE TO THE NEXT.

IF AN ENTIRE STATEMENT HAS BEEN ENTERED AND IS PARSED CORRECTLY, "OL2/PARSER" WILL DISPLAY

```
STATEMENT PARSES AS <STATEMENT TYPE>--ANY ERRORS NOTED BELOW
PARSE COMPLETE
GO AHEAD :
```

YOU SHOULD NOW ENTER YOUR NEXT STATEMENT.

IF THE STATEMENT IS INCOMPLETE AND PARSES CORRECTLY TO THE END OF WHATEVER HAS BEEN ENTERED, "OL2/PARSER" WILL DISPLAY

```
STATEMENT PARSES AS <STATEMENT TYPE>--ANY ERRORS NOTED BELOW
CONTINUE :
```

YOU SHOULD NOW ENTER THE REMAINDER OR UP TO 74 MORE CHARACTERS OF THE STATEMENT.

"OL2/PARSER" WILL DISPLAY THE MESSAGE

```
CONTINUE :
```

AS LONG AS IT FINDS NO ERRORS IN INCOMPLETE STATEMENTS. IT MAY CONSIDER A STATEMENT TO BE INCOMPLETE THAT YOU THINK IS COMPLETE. WHEN THIS OCCURS, USUALLY THERE IS NO TERMINAL SEMICOLON OR THE SEMICOLON HAS BEEN PARSED AS A NONTERMINAL SEMICOLON, AND "OL2/PARSER" EXPECTS MORE DATA.

THIS FEATURE ALSO ALLOWS YOU TO ENTER BLANK LINES FOR THE PURPOSE OF EDITING YOUR PRINTED OUTPUT WITHOUT CAUSING UNNECESSARY ERRORS. A STRING OF BLANKS WILL NOT INITIATE ANY PARSE, BUT WILL CAUSE THE PARSER TO WRITE THE MESSAGE "CONTINUE :".

IF "OL2/PARSER" DETECTS AN ERROR IN THE INPUT STRING, IT WILL DISPLAY

```
STATEMENT PARSES AS <STATEMENT TYPE>--ANY ERRORS NOTED BELOW
    <
<SYNTACTIC UNIT>: <ERROR MESSAGE>
```

```
<SYNTACTIC UNIT>: <ERROR MESSAGE>
PARSE INCOMPLETE                        %USUALLY, ALTHOUGH SOMETIMES THE
                                         MESSAGE "PARSE COMPLETE" WILL
                                         APPEAR.
GO AHEAD :
```

WHERE THE "<" CHARACTER IS DISPLAYED UNDER THE FIRST CHARACTER OF THE INPUT STRING NOT SCANNED, AND THE <SYNTACTIC UNIT>'S AND <ERROR MESSAGE>'S REFER TO THE SYNTAX OF OL/2 AS DEFINED IN "OL2/SYNTAX".

USUALLY THE OCCURENCE OF AN ERROR WILL CAUSE THE PARSE TO BREAK DOWN WITHOUT REACHING A TERMINAL NODE OF THE PARSING TREE. IN THIS CASE, THE MESSAGE "PARSE INCOMPLETE" IS DISPLAYED.

THE LIST OF ERROR MESSAGES WILL INDICATE THE PATH TAKEN DOWN THE PARSING TREE FOR THE INPUT STRING. THE ERROR MESSAGES WILL BE AS SPECIFIC AS THE SYNTAX OF OL/2 PERMITS. WHENEVER SEVERAL ALTERNATIVES OCCUR WITHIN A SYNTACTIC UNIT, ONLY GENERAL MESSAGES ARE POSSIBLE. REFER TO THE LISTING OF "OL2/SYNTAX". GENERALLY, THE FIRST MESSAGE WILL BE THE MOST USEFUL.

THE ERROR CAN ALWAYS BE FOUND IN THE ELEMENT IMMEDIATELY PREDEDING THE ERROR MARKER ('<'), UNLESS AN EARLIER ENTRY FORCED "OL2/PARSER" DOWN AN INCORRECT BRANCH OF THE PARSING TREE. (FOR EXAMPLE, TYPING 'SET' INSTEAD OF 'LET' FOR A <NEWOL2BLOCK> STATEMENT.)

IF THE MESSAGE

```
STATEMENT PARSES AS <STATEMENT TYPE>--ANY ERRORS NOTED BELOW
```

DOES NOT APPEAR, OR A NOT-ARITHMETIC STATEMENT CAUSES MESSAGES REFERRING TO <OL2LEFTHANDSIDE>, THEN MOST PROBABLY THE STATEMENT KEYWORD IS MISSING OR MISSPELLED. STATEMENT KEYWORDS, WHICH ARE UNIQUE TO THE VARIOUS STATEMENT TYPES, ARE

LET, DEFINE, DENOTE, PARTITION, SET, INTERCHANGE, IF, FOR, WHILE, UNTIL, INPUT, OUTPUT, PRINT, NODE, TRACE, PROCEDURE, PROC, END.

"OL2/PARSER" ALSO PRODUCES A LISTING OF ENTERED STATEMENTS, WITH INCORRECT STATEMENTS FLAGGED TO PROVIDE YOU WITH A RECORD OF YOUR WORK. IT IS PRINTED AFTER YOU ENTER THE WORD 'FINISH' TO TERMINATE "OL2/PARSER". IF YOU DO NOT DESIRE THE LISTING, ENTER

```
-SHIFT-N- DS
```

AS YOUR LAST INPUT STRING. IT WILL CAUSE THE JOB TO BE ABORTED.

*** REMEMBER THAT CAREFUL TYPING WILL ELIMINATE MOST ERRORS ***

BIBLIOGRAPHIC DATA SHEET

1. Report No.
UIUCDCS-R-72-504

2.

3. Recipient's Accession No.

4. Title and Subtitle
AN INTERACTIVE SYNTAX ANALYZER

5. Report Date
January, 1972

6.

7. Author(s)
WAYNE C. SANFORD

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
Dept. of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.
US NSF-GJ-328

12. Sponsoring Organization Name and Address
National Science Foundation
Washington, D. C. 20550

13. Type of Report & Period Covered

14.

15. Supplementary Notes

16. Abstracts

This report is concerned with the design of an interactive syntax analyzer for the OL/2 language. The main purpose of the syntax analyzer is to allow a user to construct statements on-line and have the analyzer identify any syntatic errors before proceeding to the next statement. This allows a program to be written which is free of all syntatic errors. This is accomplished without actually compiling any code and implemented using a compiler-compiler. It is clear that this approach to error detection allows a syntatically correct source program to be passed to a production compiler. Other applications and extensions of interactive syntax analyzers are also suggested in this report.

17. Key Words and Document Analysis. 17a. Descriptors

Interactive syntax analyzer, error correction, compiler-compiler applications.

7b. Identifiers/Open-Ended Terms

7c. COSATI Field/Group

8. Availability Statement
Unlimited

19. Security Class (This Report)
UNCLASSIFIED

20. Security Class (This Page)
UNCLASSIFIED

21. No. of Pages
93

22. Price